M U N I X
BASICS, OPTIONS
VOLUME III

D930100V3e0484 Version 1.5/02

Information in this document is subject to change without notice and does not represent a commitment on the part of Periphere Computer Systeme GmbH. The software described in this document is furnished under a license agreement. The software may be used or copied only in accordance with the terms of the agreement.

Doc.-No.: D930100V3e0484

Trademarks: MUNIX for PCS
DEC, PDP for DEC
UNIX for Bell Laboratories

MUNIX III	BASICS, OPTIONS
(binder 4)	Current MUNIX Commands - Summary
	Setting up MUNIX V1.5
	UNIX Programming
	Assembler Reference Manual
	Assembler 68000 User's Guide - outdated
	A Portable Fortran 77 Compiler
	A Tutorial Introduction to ADB
	CADMUS Testmonitor V1.0
	Bad Block Behandlung / Bad Sector Handling
	Typing Documents on the UNIX System
	Using the -MS Macros
	The UNIX Time-Sharing System
	UNIX Implementaion
	The UNIX I/O System
	Display Editing with VI
	Edit: A Tutorial
	Ex Reference Manual
Options:	MED, PASCAL
	Berkeley Extension Package

MUNIX - Volume III

BASICS,OPTIONS

Current MUNIX Commands - Summary

Setting up MUNIX V1.5

UNIX Programming

Assembler Reference Manual
Assembler 68000 User's Guide - outdated

A Portable Fortran 77 Compiler

A Tutorial Introduction to ADB

CADMUS Testmonitor V1.0

Bad Block Bchandlung
Bad Sector Handling

Typing Documents on the UNIX System
Using the MS Macros

The UNIX Time-Sharing System

UNIX Implementaion

The UNIX I/O System

Display Editing with VI

Edit: A Tutorial

Ex Reference Manual

Options:

MED - MUNIX Editor

MUNIX - PASCAL - Package

Berkeley Extension:

Berkeley Extension Summary

Writing Papers with NROFF using ME
-ME Reference Manual

Writing Tools
The Style and Diction Programms

Berkeley Font Catalogue

Screen updating and Cursor Movement

CURRENT MUNIX COMMANDS

- 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.

DEVNM — Give the device name where a specified subdirectory is resident.

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.

WHODO — Tell who is doing what on the system.

FINGER — (*BE*) List the current users including login name, terminal name, login time...

W — (*BE*) Print a summary of the current activity on the system, including what each user is doing.

USERS — (*BE*) Print a compact list of users who are on the system.

UPTIME — (*BE*) Show how long system has been up.

ID — Print user and group IDs and names.

LOGNAME — Print login name.

UNAME — Print the current system name of UNIX.

TTY — Print name of your terminal.

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.

PWD — Print name of your working directory.

CRON — Clock daemon.

1.8. Backup and Maintenance

CLRI — Clear i-node.

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.

TAR — Manage file archives on magnetic tape.

- Collect files into an archive.
- Update DECtape archive by date.
- Replace or delete DECtape files.

- Print table of contents.
- Retrieve from archive.

DUMP	(-O-) Dump the file system stored on a specified device, selectively by date, or indiscriminately. A multi-volume dump (i.e. on floppies) is possible.
RESTOR	Restore a dumped file system, or selectively retrieve parts thereof.
VOLCOPY	(-O-) Copy a file system with label checking.
LABELIT	(-O-) Create initial labels for disk or tape file systems.
SU	Temporarily become the super user with all the rights and privileges thereof. Requires a password.
FSCK	(-B-) File system consistency check and interactive repair.
FSDB	File system debugger. Used to patch up a damaged filesystem after a crash.
DCHECK	Read the directories in a file system and compare the link-count in each i-node with the number of directory entries by which it is referenced.
ICHECK	Check the number of used and free blocks. List the number of regular files, of directories and special files. If specified, a new free list is constructed.
NCHECK	Generate a pathname vs. i-number list.
CLRI	Peremptorily expunge a file and its space from a file system. Used to repair damaged file systems.
INSTALL	Install commands.
SYNC	Force all outstanding I/O on the system to completion. Used to shut down gracefully.
SHUTDOWN	(*BE*) Close down the system at a given time. Used to notify users nicely when the system is shutting down.
BADSECT	(*BE*) Create files to contain bad sectors. Less general than bad block forwarding, but better than nothing.
SCRIPT	(*BE*) Make a typescript of everything printed on the terminal during a session.
DMESG	(*BE*) Look in a system buffer for recently printed diagnostic messages and print them on the standard output.

1.9. Accounting

The UNIX System-V Accounting provides methods to collect per-process resource utilization data, record connect sessions, monitor disk utilization, and charge fees to specific logins. A set of C language programs and shell procedures is provided to reduce this accounting data into summary files and reports.

At process termination, the UNIX system kernel writes one record per process to a distinct file.

The LOGIN and INIT programs record connect sessions by writing records into /etc/wtmp. Date changes, reboots, and shutdowns are also recorded in this file.

The following list describes programs and shell-procedures of the System V accounting system:

ACCTCMS	(*ACC*) Command summary from per process accounting records.
ACCTDISK	(*ACC*) Converts user ID, login name and number of disk blocks to total accounting records.
ACCTDUSG	(*ACC*) Breaks down disk usage by LOGIN.
ACCTON	(*ACC*) Turns process accounting off.
ACCTMERG	(*ACC*) Merge or add total accounting files.
CHARGEFEE	(*ACC*) Charge a number of units to a login-name.
CKPACCT	(*ACC*) Check total size of the accounting files, initiated by the clock daemon.
MONACCT	(*ACC*) Creates summary files monthly.
PRDAILY	(*ACC*) Format a report of the previous day's accounting data.
RUNACCT	(*ACC*) Performs daily accumulation of connect, process, fee, and disk accounting.

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.
RMAIL	(*UUCP*) Restricted version of MAIL for UUCP.
BIFF	(*BE*) Be notified if mail arrives and who it is from. (not longer supported)
FROM	(*BE*) Show who is the sender of my mail.
PRMAIL	(*BE*) Print the mail which waits for you, or a specified user, in the 'post office'.
LEAVE	(*BE*) Remind you when you have to leave.
CALENDAR	Automatic reminder service for events of today and tomorrow.
WRITE	Establish direct terminal communication with another user.
MESG	Inhibit receipt of messages from WRITE and WALL.
MSGS	(*BE*) Read system messages. These messages are sent by mailing to the login 'msgs'.
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 — File transfer between CPU's.

- Automatic queuing until line becomes available and remote machine is up.
- Copy between to remote machines.
- Differences, mail, etc., between two macines.

UULOG — (*UUCP*) Maintain a summary log of UUCP and UUX transactions.

UUNAME — (*UUCP*) List the UUCP names of known systems.

UUSTAT — (*UUCP*) UUCP status inquiry and job control. Display status of, or cancel, previously specified UUCP commands, or provide general status on UUCP connections to other systems.

UUSUB — (*UUCP*) Define and monitor a UUCP subnetwork.

UUTO — Public CPU to CPU command execution. Gather files from various CPUs, execute a command on a specified CPU, and send standard output to a file on a specified CPU.

UUPICK — Accept or reject the files sent by UUTO. Looks somewhat like MAIL.

EXCR — (*NC*) Run a program on another system.

CHECKNEWS — (*UUCP*) Check to see if user has news.

INEWS — (*UUCP*) Submit news articles.

NEWS — (*UUCP*) Print news items.

POSTNEWS — (*UUCP*) Submit news articles.

READNEWS — (*UUCP*) Read news articles.

RECNEWS — (*UUCP*) Receive unprocessed articles via mail.

RJESTART — (*RJE*) RJE-status report and interactive status console.

SEND — (*RJE*) Gather files and submit RJE-jobs.

UUX — (*UUCP*) Unix to unix command execution.

UUCLEAN — (*UUCP*) Uucp spool directory clean-up.

WALL — Write to all users.

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.

A68 — A fast M68000-Assembler,

- Creates object program consisting of
  code, possibly read-only
  initialized data
  uninitialized data.
- Object code normally includes a symbol table.
-

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.

CURSES — (*BE*) Library of screen functions which allows screen updating (with user input) and cursor motion optimization.

ADB — (-O-) Interactive debugger.

- Postmortem dumping.

Examination of arbitrary files, with no limit on size.

Interactive breakpoint debugging with the debugger as a separate process.

Symbolic reference to local and global variables.

Stack trace for C programs.

Output formats:
  1-, 2-, or 4-byte integers in hex, octal
  or decimal
  single and double floating point
  character and string
  disassembled machine instructions

Patching.

Searching for integer, character, or floating patterns.

- 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.

- Range of dumping is controllable.

XD — Hexadecimal file dump.

LD — Link edit. Combine relocatable object files. Insert required routines from specified libraries.

	Resulting code may be sharable. 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. • 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. • Knows about CC, PASCAL, YACC, LEX etc.
ERROR	(*BE*) Analyze and disperse compiler error messages. • Knows about error messages produced by MAKE, CC, CPP, AS, LD, LINT, PC, F77. Attempts to determine which language processor produced each error message. Determines source file and line number to which the error message refers. Determines if the error message is to be ignored, or not. • Inserts the error message into the source file as comment
HELP	Ask for help.
NOTES	A news system.
PACK	(-V7-) Packs or unpacks many files <---> one.
RANLIB	(-V7-) Convert archives to random libraries.
REGCMP	Regular expression compile.
TIMEX	Time a command; report process data and system activity.

1.12. UNIX Programmer's Manual

MANUAL — Machine-readable version of the UNIX Programmer's Manual.

- System overview.
- All commands.
- All system calls.
- All subroutines in C and assembler libraries.
- All devices and other special files.
- Formats of file system and kinds of files known to system software.
- Boot and maintenance procedures.

MAN — Print specified manual section on your terminal.

MAN — (*BE*) Give information from the on line programmers' manual.

- Gives all commands whose description contains any of a specified set of keywords.

  Attempts to locate manual sections related to a specified list of files.
- Formats a specified set of manual pages.

CATMAN — (*BE*) Create the preformatted versions of the on-line manuals.

APROPOS — (*BE*) Show which manual sections contain instances of any of the given keywords in their title.

WHATIS — (*BE*) Look up a given command and give the header line from the manual section.

1.13. Computer-Aided Instruction

LEARN — A program for interpreting CAI scripts, plus scripts for learning about UNIX by using it.

- 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 — (-O-) 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.

- General purpose language designed for structured programming.

  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.

  Operations intended to give machine-independent control of full machine facility, including to-memory operations and pointer arithmetic.

  Macro preprocessor for parameterized code and inclusion of standard files.
- All procedures recursive, with parameters by value.
- Machine-independent pointer manipulation.
- Object code uses full addressing capability of the M68000.
- Runtime library gives access to all system facilities.
- Floating point arithmetic realized by software.
- Definable data types.
- 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.

- Full cross-module checking of separately compiled programs.

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

CPP — The C language preprocessor. Also used by F77 and Pascal.

INDENT — (-V-) Indent and format a C program source.

2.2. Fortran

F77 — A full compiler for ANSI Standard Fortran 77.

- Compatible with C and supporting tools at object level.
- Optional source compatibility with Fortran 66.
- Free format source.
- Optional subscript-range checking, detection of uninitialized variables.

  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 like C to Fortran.

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

EFL — (-V7-) Extended Fortran Language.

2.3. Pascal

PC (*PAS*) Compile and/or link programs in the PASCAL 68000 language.
PASCAL 68000 is an extended implementation of the PASCAL language. Specifically PASCAL 68000 complies almost completely with the requirements of the ISO standard proposal for PASCAL. Some of the features of PASCAL 68000 are

- General purpose language
- Block oriented language
- Strong type checking
- Variety of data structures:
  simple types, arrays, records,
  sets, files, pointers, strings

  Conformant arrays

  Various control statements

  Predefined procedures and functions

  Seperate compilation of modules

  Import and export of variables,procedures and functions

  Linking C, FORTRAN or assembler modules to PASCAL 68000 modules

2.4. Snobol

SNO — (*SNO*) Is an implementation of the SNOBOL language (especially the SPITBOL version). Its unusual support for string, list and table manipulation makes SNO a powerful tool for several applications.

- SNO has some of the best features of Basic and Lisp: It is interactive, has 'rubber memory' for strings, lists and associative tables, and finally it is easy to learn.

SNO is qualified for the following applications:
text editing jobs
small interactive data bases
small translators: mini-languages, macros...

Preprocessor snif

2.5. Other Algorithmic Languages

BS — A compiler/interpreter for modest-sized programs. A descendant of Basic and SNOBOL 4 with a little C language thrown in.

- Statements from console execute immediately.
- Statements from a file compile for later execution (by default).
- Line-at-a-time debugging.
- Many builtin functions.

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

- Unlimited precision decimal arithmetic.
- Appropriate treatment of decimal fractions.
- Arbitrary input and output radices, in particular binary, octal, decimal and hexadecimal.

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.

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

BASIC — Basic Interpreter.

FACTOR — (-V-) Factor a number.

IFPROLOG ("PRO") The prolog interpreter system.

2.6. Macroprocessing

M4 A general purpose macroprocessor.

- Stream-oriented, recognizes macros anywhere in text.
- Syntax fits with functional syntax of most higher-level languages.

  Can evaluate integer arithmetic expressions.

2.7. 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.

- BNF syntax specifications.
- Precedence relations.
- 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.

- Full regular expression, plus left and right context dependence.

  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.

- 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.
- Add, delete, change, copy, move or join lines.
- Permute or split contents of a line.
- Replace one or all instances of a pattern within a line.

  Combine or split files.

  Escape to Shell (command language) during editing.

  Do any of above operations on every pattern-selected line in a given range.
- Optional encryption for extra security.

MED (*MED*) This editor allows you to edit a file using the screen and the cursor keys somewhat like paper, pencil and eraser.

- Add, delete, change, copy lines.
- Split, concatenate lines.
- Find lines by number or pattern.

  Manage previous defined rectangles.

  Switch to another file.

  Macro facility.

  Install windows for working simultaneously with different files.
- Escape to Shell during editing.

EX (*BE*) The line oriented text editor EX is a superset of the ED editor from UNIX V7 and the root of the interactive display function VIEW and the family of editors: EX, EDIT, VI.

- Find lines by number or pattern.
- Add, delete, change, copy, move or join lines.

  Permute or split contents of a line.

  Replace one or all instances of a pattern within a line.

  Combine or split files.

  Switch to the location of a 'tag'.

  Enter intraline editing.

  Reverse the effects of the last command.

  Escape to Shell during editing.

  Indent automatically.
- Define abbreviations.
- Attempt to recover the buffer in case of hangups or crashes.

- Read and execute commands from a specified file.
- Simulate an intelligent terminal on a dumb terminal.

EDIT — (*BE*) A small version of EX. Avoids some of the complexities of EX to provide an environment for new and casual users. (not longer supported)

- Find lines by number or pattern.
- Add, delete, change, copy or move lines.
- Replace a pattern in a line.
- Add the contents of a file.
- Reverse the effects of the last command.
- Escape to Shell.
- Attempt to recover the buffer in case of hangups or crashes.

VI — (*BE*) The screen oriented editor VI is based on EX (see above). Additional attributes:

- Numerous commands for file manipulation.
  i.e. edit file containing the tag 'tag'
  at the first line of 'tag'

  Extensive command set for scrolling, paging and cursor motion.
  i.e. move to the end of line
  move to the begin of the next word

  Various units of text can be handled: words, sentences, sections.
  i.e. duplicate sentence
  delete word

  Searching for strings by a set of different conditions.
  i.e. matches any character between 'x' and 'y'
  matches the end of a word
- Definition of macros for saving time by typing commands.
- Escape to the line oriented editor EX.

VIEW — (*BE*) Interactive display function. Works like the VI - but with read-only files. (not longer supported)

CTAGS — (*BE*) Make a tags file for EX from the specified C, PASCAL and FORTRAN sources. A tags file gives the locations of specified objects (in this case functions) in a group of files.

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.

- 25,000-word list includes proper names.
- Handles common prefixes and suffixes.
- Collects words to help tailor local spelling lists.

CRYPT — Encrypt and decrypt files for security.

HYPHEN — Find hyphenated words.

MKSTR — (*BE*) Used to create a file of error messages by massaging C source code.

Places all error messages from a C source file in a specified file.

Keys on the string 'error("' to process the error messages to the message file.

The copy of the C source file contains pointer into the message file to retrieve the error message.

XSTR (*BE*) Extract strings from C programs to implement shared constant strings.

- Maintains a file into which strings of component parts of a large program are hashed.

  The strings are replaced with references to the common area.

DICTION (*BE*) Find wordy sentences in a document.

- Finds all sentences that contain phrases from a data base of bad or wordy diction.
- The user may supply his own pattern file.

EXPLAIN (*BE*) Interactive thesaurus for the phrases found by diction.

STYLE (*BE*) Analyze surface characteristics of a document.

- Reports on
  - readability
  - sentence length and structure
  - word length and usage
  - verb type
  - sentence openers
- Options to locate sentences with certain characteristics.

CUT (-V-) Cut out selected fields of each line of a file.

DBADD (*EMAC*) Add entry to an Emacs data base.

EMACS (*EMAC*) A screen editor.

PREP Prepare text for statistical processing.

REFER Find and insert literature references in documents.

3.2. Document Formatting

TROFF

NROFF — Advanced typesetting. TROFF drives a Graphic Systems phototypesetter; NROFF drives ascii terminals of all types. TROFF and NROFF style is similar to ROFF (not available on MUNIX), but they are capable of much more elaborate feats of formatting, when appropriately programmed. TROFF and NROFF accept the same input language.

- All ROFF capabilities available or definable.
- Completely definable page format keyed to dynamically planted "interrupts" at specified lines.

  Maintains several separately definable typesetting environments (e.g., one for body text, one for footnotes, and one for unusually elaborate headings).

  Arbitrary number of output pools can be combined at will.

  Macros with substitutable arguments, and macros invocable in mid-line.

  Computation and printing of numerical quantities.

  Conditional execution of macros.

  Tabular layout facility.

  Positions expressible in inches, centimeters, ems, points, machine units or arithmetic combinations thereof.

  Access to character-width computation for unusually difficult layout problems.

  Overstrikes, built-up brackets, horizontal and vertical line drawing.

  Dynamic relative or absolute positioning and size selection, globally or at the character level.

  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 and TBL are fully compatible with TROFFand NROFF.

MS — The standardized manuscript layout package of V7 for use with NROFF or TROFF. This document was formatted with MS.

Page numbers and draft dates.

Automatically numbered subheads.

Footnotes.

- Single or double column.
- Paragraphing, display and indentation.
- Numbered equations.

MM — The standardized manuscript layout package of System III for use with NROFF or TROFF. Some features different from the MS macro package:

- Table of contents.

Static and Floating displays.

Special formatting macros for preparing memoranda and released papers..

ME — (*BE*) Package from Berkeley 4.1 bsd for formatting technical papers with NROFF or TROFF. Easy to learn.

COLCRT — (-V-) Filter nroff output for CRT previewing.

MMCHEK — (-V-) Check usage of mm macros and eqn delimiters.

MMT — (-V-) Typeset documents, view graphs, and slides.

VTROFF — (*BE*) Troff for raster printer/plotter.

LTROFF — (*TSP*) Troff for CANON laser beam printer. Ltroff accepts the same input language as troff.

EQN — A mathematical typesetting preprocessor for TROFF. Translates easily readable formulas, either in-line or displayed, into detailed typesetting instructions.

- Automatic calculation of size changes for subscripts, sub-subscripts, etc.

Full vocabulary of Greek letters and special symbols, such as 'gamma', 'GAMMA', 'integral'.

Automatic calculation of large bracket sizes.

Vertical "piling" of formulae for matrices, conditional alternatives, etc.

Integrals, sums, etc., with arbitrarily complex limits.

Diacriticals: dots, double dots, hats, bars, etc.

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.

- 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.

- Computes column widths.

Handles left- and right-justified columns, centered columns and decimal-point alignment.

- Places column titles.
- Table entries can be text, which is adjusted to fit.
- Can box all or parts of table.

GREEK — Fancy printing on Diablo-mechanism terminals like DASI-300 and DASI-450, and on Tektronix 4014.

- Gives half-line forward and reverse motions.

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.

VFONTINFO — (*BE*) Inspect and print out information about unix fonts.

SOELIM — (*BE*) Eliminate .so's from nroff input.

CHECKNR — (*BE*) Check NROFF/TROFF files.

- Knows about MS and ME macro packages.
- Checks unknown commands.
- Checks mismatched opening and closing delimiters in case of
  - macros which always come in pairs
  - font changes
  - size changes

PTI — (*BE*) Interpret a stream from the standard output of TROFF as it would act on the typesetter.

FMT — (*BE*) Simple text formatter.

- Produces an output with lines as close to 72 characters as possible.

Spacing at the beginning of input lines and blank lines are preserved.

4. Information Handling

SORT — Sort or merge ASCII files line-by-line. No limit on input size.

- Sort up or down.
- Sort lexicographically or on numeric key.
  Multiple keys located by delimiters or by character position.
  May sort upper case together with lower into dictionary order.
- 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.

- Publish lines that were originally unique, duplicated, or both.
- May give redundancy count for each line.

TR — Do one-to-one character translation according to an arbitrary code.

- May coalesce selected repeated characters.
- May delete selected characters.

DIFF — Report line changes, additions and deletions necessary to bring two files into agreement.

- May produce an editor script to convert one file into another.
- A variant compares two new versions against one old one.

SDIFF — Produce a side-by-side listing of two files indicating those lines that are different.

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.

- May print all lines that fail to match.
- May print count of hits.
- 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.

- Lines may be selected by address or range of addresses.
- Control flow and conditional testing.
- Multiple output streams.
- 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.

Patterns include regular expressions, arithmetic and lexicographic conditions, boolean combinations and ranges of these.

- Data treated as string or numeric as appropriate.
- Can break input into fields; fields are variables.
- Variables and arrays (with non-numeric subscripts).
- Full set of arithmetic operators and control flow.
- Multiple output streams to files and pipes.
- Output can be formatted as desired.
- Multi-line capabilities.

BFS — Big file scanner. Similar to ED, except it is read-only and processes larger files. Useful for identifying sections of a large file where CSPLIT can be used to divide it.

- Maximum file size is 1024-K bytes.
- Scans actual file, not a copy.
- All ED address expressions are supported.
- Regular expression processing.
- Most ED commands operate.
- Many additional commands.
-

PWCK — Scan the password file and note inconsistencies..

GRPCK — Verify entries in the group file.

CREF — Make a cross-reference listing of an assembler or C program.

XREF — (*PAS*) Create a cross reference listing from a C or Pascal program.

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.

TPLOT — A set of filters for printing graphs produced by GRAPH and other programs on various terminals. Filters provided for Tektronix 4014, DASI terminals, Versatec printer/plotter.

PLOT — Graphics filters.

TC — (-V7-) Phototypesetter simulator.

TK — (-V7-) Paginator for the Tektronix 4014.

6. Source Code Control System SCCS

SCCS is a collection of commands for controlling changes to files of text (typically, the source code of programs or text of documents).

ADMIN	(*SCCS*) Create new SCCS files and change parameters of existing ones.
CDC	(*SCCS*) Change the delta commentary of an SCCS delta.
COMB	(*SCCS*) Combine SCCS deltas.
DELTA	(*SCCS*) Make a delta (change) to an SCCS file.
GET	(*SCCS*) Create an ASCII text file
PRS	(*SCCS*) Print an SCCS file.
RMDEL	(*SCCS*) Remove a delta from an SCCS file.
SACT	(*SCCS*) Inform the user of any impending deltas to a named SCCS file.
SCCSDIFF	(*SCCS*) Compare two versions of an SCCS file and generate a list of differences.
UNGET	(*SCCS*) Undo a previous GET of an SCCS file.
VAL	(*SCCS*) Determine if a specified file is an SCS file meeting characteristics specified by the argument list.
WHAT	(*SCCS*) Identify SCCS files.

7. Novelties, Games, and Things That Didn't Fit Any Where Else

ARITHMETIC	Speed and accuracy test for number facts.
BACKGAMMON	A player of modest accomplishment.
BANNER	Print output in huge letters.
BCD	Converts ascii to card-image form.
CAL	Print a calendar of specified month and year.
FCOOKIE	Presents a random fortune cookie on each invocation. Limited jar of cookies included.
QUIZ	Test your knowledge of Shakespeare, Presidents, capitals, etc.
STARTREK	Strategy game. Destroy the klingons.
UNITS	Convert amounts between different scales of measurement. Knows about hundreds of units. For example, how many km/sec is a parsec/megayear?
WUMP	Hunt the wumpus, thrilling search in a dangerous cave.
FISH	(*E*) Childrens' card guessing game. (not longer supported)
HANGMAN	
HANG	(*BE*) Word guessing games. Uses the dictionary supplied with SPELL.
TWINKLE1	
TWINKLE2	(*BE*) Milky way on the screen. (not longer supported)
WORM	(*BE*) Lead the worm to random points. (not longer supported)

WORMS	(*BE*) Several worms running around on screen. (not longer supported)
BACK	The game of backgammon.
BJ	The game of black jack.
CRAPS	The game of craps.
MOO	Guessing game.
REVERSI	Reversi, a game of dramatic reversals.
ROGUE	Exploring The Dungeons of Doom - the biggest game you 've ever seen.
TTT	Tic-tac-toe.
WUMP	The game of hunt-the-wumpus.
HUP	(-O-) Ring the terminal bell. (not longer supported)
TRUE	
FALSE	Provide truth values.

Setting up MUNIX

Version 1.5

Trademarks:

MUNIX, CADMUS	for PCS
DEC, PDP	for DEC
UNIX	for Bell Laboratories

Setting up MUNIX Version 1.5 / 02

Unix grows at a considerable speed, getting more and more complex. Moreover, systems by PCS come in a wide range of hardware configurations. These factors together make the setting up process a nontrivial task. It is complicated to describe this process in a linear fashion. We hope we made the process a bit easier to follow by writing it down as an "algorithm" with conditionals and (phooey) gotos.

ENTRY

If you have a new system, where the disks have been configured for 1.5 by PCS, go to label *NEWSTAND.*

If you have an old system with pre 1.5 software, go to *SAVEOLD*, else go to *READV1.5*

SAVEOLD

Save your own files on a backup medium. When you read the files in later, you will read them probably onto a 1024 byte filesystem. In addition, save several system files. The first set of system files should be saved for later examination. They will not be copied back! The second set is saved and later copied back.

The following files are in the first set, which is not copied back, but must be inspected, if you have introduced local additions or changes.

File	Reason
/.profile	
/etc/fstab	your file system names
/etc/group	your groups
/etc/keycap	your keyboard capabilities
/etc/passwd	your user entries
/etc/profile	
/etc/rc	
/etc/termcap	your terminal capabilities
/etc/ttys	your terminal names
/etc/ttytype	your terminal types and speeds
/usr/lib/crontab	your local additions
/usr/sys/conf.h	

The following files must be saved to be copied back later:

File	Reason
/tmp/spool/*	NEWS
/usr/lib/news/active	NEWS
/usr/lib/news/history	NEWS
/usr/lib/news/history.dir	NEWS
/usr/lib/news/history.pag	NEWS
/usr/lib/news/net.recording	NEWS
/usr/lib/news/recording	NEWS
/usr/lib/news/sys	NEWS
/usr/lib/uucp/L-devices	UUCP
/usr/lib/uucp/L.sys	UUCP
/usr/local/*	your own commands

If you had other local commands in /bin or /usr/bin, move them to /usr/local!

READV1.5
If the new release is on a set of floppies, go to *READFLOPPY*.

READTAPE
You have to read the new version from tape or cartridge. You got two or more tapes or streamer cartridges. The first tape or cartridge contains the standalone programs and a minimal root file system in dump format (see *dump* (5)), the other tape(s) or cartridge(s) contain additional files in cpio format. The first tape or cartridge starts with four standalone programs, i.e. boot (a boot program), check (a disk formatter program), mkfs (to make a new filesystem), and restor (to restore from a dump tape). A dump of a minimal root file system follows. Additional filesets are on the second tape.

You start by loading the boot program from the tape or streamer cartridge. After that, the boot program can be used to load more programs from the tape or other medias. The minitor must prompt you with a ".".

Power on (if you want to boot from tape, set the magtape drive to ONLINE)

hit any key
the Minitor prompts with
what ?

Your input	Machine Reaction
rt or **rs**	select tape or streamer
/	reads in first file (boot program)
g	starts boot program, prompts for new program

If your disks are not hardware formatted, or if they have not been checked for bad blocks, go to *FORMAT*, else go to *MKFS*

FORMAT

The second program on the tape, check, formats the disks and checks them for bad blocks. If bad blocks are found, they are recorded in a bad sector table, together with a replacement block. Thus, the disk appears 100% ok.

Your Input	Machine Reaction
tm(0,1) or st(0,1)	load second file (check) from tape or streamer check program starts

The check program will ask for a device name. You enter rl, hl, rm or hk, and the disk number in brackets. The disk number is the physical disk number, as interpreted by the controller. You may e.g. have a Fujitsu 80 Mb disk, which is "seen" by the controller as three physical disks, 2 RK07s and 1 RK06. So you would enter the names hk(0), hk(1) and hk(2). The disk number must not be confused with major or minor device numbers, or with device names. /dev/hk1 is not the same as hk(1) for the check program!

Then you are asked for a command. You enter "f" to format the disk, "b" to make a bad sector scan, and "q" to quit. You repeat this sequence for all disks, and then enter "exit" to exit the check program.

MKFS

The third program on the tape is the mkfs program. It is used to make a new root file system. The boot program is still loaded and prompts you for a new program:

Your input	Machine Reaction	Comment
tm(0,2) or st(0,2)	loads mkfs program	
	mkfs prompts with "file sys size:"	
8000 or 9600 or 9636		enter root fs size for rl, rm or hk resp.
	mkfs prompts with "file system?"	
rl(0,0) or rm(0,0) or hk(0,0)		enter your type of root device

8000 blocks are used for RL, 9600 for RM and 9636 for HK.

RESTOR

Now you load the restor program, and read the root file system onto the newly formatted disk.

hit INIT key
the Minitor prompts with

Your input	Machine Reaction	Comment
rt or rs	select tape or streamer	
/	reads in first file (boot program)	
g	starts boot program, prompts for new program	
tm(0,3) or st(0,3)	loads restor program	
	restor prompts "Tape?"	
tm(0,4) or st(0,4)		the root file system is the 4th file on the tape
	restor prompts "Disk?"	
rl(0,0) or rm(0,0) or hk(0,0)		enter the type of disk for the root file system
	restor prompts "last chance before scribbling on disk"	
Hit <Return>	a few seconds will pass before the tape or the streamer cartridge moves	wait until restor prompts with "Exit called"

go to *LOADAUNIX*.

READFLOPPY

You have a bunch of floppies, which must be read in one by one. One floppy contains the standalone programs, the others contain a root filesystem in dump format and additional files in cpio format for the usr filesystem. The disk is emulated as an RL02. Insert the floppy with standalone programs into drive 0 and start the boot program.

Your input	Machine Reaction
rx	select floppy as boot device
/boot	reads in boot program
g	starts boot program, prompts for new program

Formatting and checking the disks is analogous to the sequence of actions described at *READTAPE*. So we just give the commands and a short description here.

Your input	Machine Reaction	Comment
	boot prompts for a program	
rx(2,0)check	load disk check/format program asks for devname(unit)	
rl(0)	check/format rl disk 0, asks for command	
f		format disk
y		answer each question with **y** for **yes**
y	formats disk,	
b	bad sector scan	
q	quit actions for rl(0)	
rl(1)	check/format rl disk 1, asks for command	
f		format disk
y		answer each question with **y** for **yes**
y	formats disk,	
b	bad sector scan	
q	quit actions for rl(1)	
exit	exit check program	
rx(2,0)mkfs	load mkfs from floppy, asks file system size	
8000		root filesystem size
rl(0,0)	makes new file system, exits	
rx(2,0)restor	load restor program, asks for "Tape"	

Change floppy to #1 of root file system

Your input	Machine Reaction
rx(2,0)	"Tape" is the floppy drive asks for "Disk"
rl(0,0)	"Disk" is the RL drive "last chance before scribbling on disk"
press "return" key	reads in first floppy, "mount volume 2..."

Change floppy to #2 of root file system and press return. Repeat this with floppy #3, #4 and so on. Eventually, the restor will be done.

LOADAUNIX

You now have a root file system. This contains a unix kernel, **/unix**, configured for your system, and a unix kernel, **/aunix**, that can run on a variety of disks. Load **/unix** with the minitor or with the boot program from the root file system and start it. If you have a minitor with autoload capability, **/unix** will be loaded automatically after hitting the **INIT** key, otherwise, follow the description below.

Your Input	Machine Reaction	Comment
Hit INIT key	Minitor prompts with "." if not, press RETURN until Minitor prompts with "."	
hk or **rm** or **rl**		select disk type (default is HK)
/unix	load /unix	
g	start /unix	

If you load **/aunix** instead of **/unix**, **aunix** will ask some questions before it comes up. When the kernel asks you whether it shall go into multi user mode, answer **n** for **no**.

Your Input	Machine Reaction
	going Multi-user mode ? (y/n):
n	
	to go multi-user type "init 2".

SETDATE

Set the correct date with the command

date *mmddhhmmyy*

mm	is the	month number
dd	is the	day number in the month
hh	is the	hour number (24 hour system)
mm	is the	minute number
yy	is the	last two digits of the year

For example: date 0415093084 sets the date to Apr 15, 1984, 9:30 pm.

This will be usefull if you want to make a new kernel.

MAKEDEV

Look at the entries in directory **/dev**. The special files in this directory should reflect your hardware configuration. If not, call **make** for the needed entries.

MKFS

Next you must make additional filesystems on the disk, first of all the system for **/usr**. The following commands make a default **/usr** filesystem:

Device	Command
RL:	mkfs /dev/rrl2 20380
HK:	mkfs /dev/rhk2 53636
RM:	mkfs /dev/rrm2 105120

For HK and RM, the logical disk partion for the swap space (minor unit 1) can be partly used for a **/tmp** filesystem, if the main memory has not more than 1 megabyte. E.g. for HK, you can use 4000 blocks of the 8910 blocks of the swap space for a **/tmp** filesystem. The 8910 blocks would be divided into the **/tmp** area from 0 to 3999, and the swap space proper from 4000 to 8909. Both the special files for /tmp (**/dev/tmp**) and for swap (**/dev/swap**) would have minor unit number 1. For swap, we would have SWPLO = 4000 instead of 0, and NSWAP = 4910 instead of 8910, in **/usr/sys/conf.h**.
After the mkfs for all filesystems, mount them on their respective directories and read the second tape or cartridge with the /usr files. Make sure you are in the / directory (**cd /**). Use cpio with the options **-iBvmd** on the tape or **-iSvmd** on the streamer or **-ivmd** on the floppy:

Device	Commands
streamer:	cd / cpio -iSvmd </dev/nrst0
tape:	cd / cpio -iBvmd </dev/nrmt0
floppy:	cd / cpio -ivmd < /dev/rrx2

Repeat as many times as there are filesets listed on your distribution media.

NEWKERNEL

You may configure a Unix kernel specially adopted to your hardware. This is best done by calling **/etc/newconf** in directory **/usr/sys**, followed by **make**. You may wish to look at the files **/usr/sys/conf.h**, **/usr/sys/c.c** and **/usr/sys/name.c** before starting **make**, and edit them if you want to deviate from the standard values supplied there. After the make, you have a new kernel **/nunix**, which you should try out, as described in *newconf(8)*. Go to *MODIFETC*

NEWSTAND

You can immediately boot **/unix**.

Power on (if you want to boot from tape, set the magtape drive to ONLINE)

hit any key
the Minitor prompts with
what ?

Your input	Machine Reaction	Comment
hk or **rm** or **rl**		select disk type (default is HK)
/unix	load /unix	
g	start /unix	

However, you should edit **/usr/sys/name.c** in order to introduce correct identifying information, followed by **make** in the directory **/usr/sys**.

In the directory **/usr/local/filesets** you will find several files with a suffix of *'list'* (e.g. rootlist). These files contain the pathnames of the files distributed with your system.
In the initial state, as shipped by PCS, the system has only the files from rootlist and usrlist copied on its disk(s). If you want to use files from additional filesets, you have to read them in from your distribution media.

To read in a new set of files, be aware of the following:

first, you have to go to **/** in all cases

```
cd /
```

if reading in filesets from tape or streamer cartridge, first rewind the tape or the streamer cartridge and then skip the filesets you do not need:

Device	Commands	Comment
tape:	< /dev/rmt0	rewind tape
	tm fsf n	skip n filesets on tape
	cpio -iBmvd < /dev/nrmt0	read a fileset without rewinding
streamer:	< /dev/rst0	rewind streamer cartridge
	stskip n	skip n filesets on cartridge
	cpio -iSmvd < /dev/nrst0	read a fileset without rewinding

Continue by alternating the skip and the read commands according to your needs. After having finished reading in the filesets you need, rewind:

Device	Commands	Comment
tape:	< /dev/rmt0	rewind tape
streamer:	< /dev/rst0	rewind streamer cartridge

if reading in from floppies, insert the floppies labeled with the fileset you need (any order), and use for each floppy the command:

```
cpio -ivmd < /dev/rrx2
```

If you wish to remove filesets from your disk(s), we suggest to use the files **/usr/local/filesets/*list** to get complete lists of related files. One way to do this is to use the shell command listed bellow:

```
sed '/^#/d' /usr/local/filesets/xxxlist | sed '/^$/d' | \
( while read F; do rm $F; done )
```

MODIFETC

There are several files, mainly in **/etc**, which have to reflect your hardware configuration, how many terminals you have, what speeds they run on, what kinds of terminals are attached and so on. For a description of the files, read manual sections 5 and 8.

You should inspect the following files and modify them if necessary:

File	Reason
/etc/bcheckrc	for definition of TZ
/etc/checklist	for your disks, used by fsck, look at old /etc/fstab
/etc/group	add your own groups, do not modify otherwise
/etc/inittab	add your terminals and types, look at old **/etc/ttys**, **/etc/ttytype**
/etc/issue	will be printed before login
/etc/passwd	add your own users, do not modify otherwise
/etc/profile	for definition of TZ, look at old /etc/profile
/etc/rc	add your own mount commands, look at old /etc/rc
/usr/lib/crontab	used by clock daemon for regular tasks

You should examine and modify the following files, if you intend to use them:

File	Reason
/etc/checkall	fsck for your disks
/etc/filesave	backup shell script
/etc/gettydefs	for other speeds
/etc/motd	message of the day
/etc/tapesave	backup shell script
/usr/lib/acct/holidays	for local holiday schedule

RESTOROLD

If you saved old files, read them in now.

ENDOFSETUP

Now all files are set up. You can go into multi user mode with **init 2**. Make sure that you can login at all terminals.

UNIX Programming

This paper is an introduction to programming on the UNIX system. The emphasis is on how to write programs that interface to the operating system, either directly or through the standard I/O library. The topics discussed include

- handling command arguments
- rudimentary I/O; the standard input and output
- the standard I/O library; file system access
- low-level I/O: open, read, write, close, seek
- processes: exec, fork, pipes
- signals – interrupts, etc.

There is also an appendix which describes the standard I/O library in detail.

UNIX Programming — Second Edition

Brian W. Kernighan

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

This paper describes how to write programs that interface with the UNIX operating system in a non-trivial way. This includes programs that use files by name, that use pipes, that invoke other commands as they run, or that attempt to catch interrupts and other signals during execution.

The document collects material which is scattered throughout several sections of *The UNIX Programmer's Manual* [1] for Version 7 UNIX. There is no attempt to be complete; only generally useful material is dealt with. It is assumed that you will be programming in C, so you must be able to read the language roughly up to the level of *The C Programming Language* [2]. Some of the material in sections 2 through 4 is based on topics covered more carefully there. You should also be familiar with UNIX itself at least to the level of *UNIX for Beginners* [3].

2. BASICS

2.1. Program Arguments

When a C program is run as a command, the arguments on the command line are made available to the function `main` as an argument count `argc` and an array `argv` of pointers to character strings that contain the arguments. By convention, `argv[0]` is the command name itself, so `argc` is always greater than 0.

The following program illustrates the mechanism: it simply echoes its arguments back to the terminal. (This is essentially the `echo` command.)

```
main(argc, argv)    /* echo arguments */
int argc;
char *argv[];
{
    int i;

    for (i = 1; i < argc; i++)
        printf("%s%c", argv[i], (i<argc-1) ?       '\n');
}
```

`argv` is a pointer to an array whose individual elements are pointers to arrays of characters; each is terminated by `\0`, so they can be treated as strings. The program starts by printing `argv[1]` and loops until it has printed them all.

The argument count and the arguments are parameters to `main`. If you want to keep them around so other routines can get at them, you must copy them to external variables.

2.2. The "Standard Input" and "Standard Output"

The simplest input mechanism is to read the "standard input," which is generally the user's terminal. The function `getchar` returns the next input character each time it is called. A file may be substituted for the terminal by using the < convention: if `prog` uses `getchar`,

then the command line

```
prog <file
```

causes `prog` to read `file` instead of the terminal. `prog` itself need know nothing about where its input is coming from. This is also true if the input comes from another program via the pipe mechanism:

```
otherprog | prog
```

provides the standard input for `prog` from the standard output of `otherprog`.

`getchar` returns the value `EOF` when it encounters the end of file (or an error) on whatever you are reading. The value of `EOF` is normally defined to be -1, but it is unwise to take any advantage of that knowledge. As will become clear shortly, this value is automatically defined for you when you compile a program, and need not be of any concern.

Similarly, `putchar(c)` puts the character `c` on the "standard output," which is also by default the terminal. The output can be captured on a file by using >: if `prog` uses `putchar`,

```
prog >outfile
```

writes the standard output on `outfile` instead of the terminal. `outfile` is created if it doesn't exist; if it already exists, its previous contents are overwritten. And a pipe can be used:

```
prog | otherprog
```

puts the standard output of `prog` into the standard input of `otherprog`.

The function `printf`, which formats output in various ways, uses the same mechanism as `putchar` does, so calls to `printf` and `putchar` may be intermixed in any order; the output will appear in the order of the calls.

Similarly, the function `scanf` provides for formatted input conversion; it will read the standard input and break it up into strings, numbers, etc., as desired. `scanf` uses the same mechanism as `getchar`, so calls to them may also be intermixed.

Many programs read only one input and write one output; for such programs I/O with `getchar`, `putchar`, `scanf`, and `printf` may be entirely adequate, and it is almost always enough to get started. This is particularly true if the UNIX pipe facility is used to connect the output of one program to the input of the next. For example, the following program strips out all ascii control characters from its input (except for newline and tab).

```
#include <stdio.h>

main()    /* ccstrip: strip non-graphic characters */
{
     int c;
     while ((c = getchar()) != EOF)
          if ((c >= ' ' && c < 0177) || c == '\t' || c == '\n')
               putchar(c);
     exit(0);
}
```

The line

```
#include <stdio.h>
```

should appear at the beginning of each source file. It causes the C compiler to read a file (*/usr/include/stdio.h*) of standard routines and symbols that includes the definition of `EOF`.

If it is necessary to treat multiple files, you can use `cat` to collect the files for you:

```
cat file1 file2     | ccstrip >output
```

and thus avoid learning how to access files from a program. By the way, the call to `exit` at the end is not necessary to make the program work properly, but it assures that any caller of the

program will see a normal termination status (conventionally 0) from the program when it completes. Section 6 discusses status returns in more detail.

3. THE STANDARD I/O LIBRARY

The "Standard I/O Library" is a collection of routines intended to provide efficient and portable I/O services for most C programs. The standard I/O library is available on each system that supports C, so programs that confine their system interactions to its facilities can be transported from one system to another essentially without change.

In this section, we will discuss the basics of the standard I/O library. The appendix contains a more complete description of its capabilities.

3.1. File Access

The programs written so far have all read the standard input and written the standard output, which we have assumed are magically pre-defined. The next step is to write a program that accesses a file that is *not* already connected to the program. One simple example is *wc*, which counts the lines, words and characters in a set of files. For instance, the command

```
wc x.c y.c
```

prints the number of lines, words and characters in `x.c` and `y.c` and the totals.

The question is how to arrange for the named files to be read — that is, how to connect the file system names to the I/O statements which actually read the data.

The rules are simple. Before it can be read or written a file has to be *opened* by the standard library function `fopen`. `fopen` takes an external name (like `x.c` or `y.c`), does some housekeeping and negotiation with the operating system, and returns an internal name which must be used in subsequent reads or writes of the file.

This internal name is actually a pointer, called a *file pointer*, to a structure which contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and the like. Users don't need to know the details, because part of the standard I/O definitions obtained by including `stdio.h` is a structure definition called `FILE`. The only declaration needed for a file pointer is exemplified by

```
FILE *fp, *fopen();
```

This says that `fp` is a pointer to a `FILE`, and `fopen` returns a pointer to a `FILE`. (`FILE` is a type name, like `int`, not a structure tag.

The actual call to `fopen` in a program is

```
fp = fopen(name, mode);
```

The first argument of `fopen` is the name of the file, as a character string. The second argument is the mode, also as a character string, which indicates how you intend to use the file. The only allowable modes are read (`"r"`), write (`"w"`), or append (`"a"`).

If a file that you open for writing or appending does not exist, it is created (if possible). Opening an existing file for writing causes the old contents to be discarded. Trying to read a file that does not exist is an error, and there may be other causes of error as well (like trying to read a file when you don't have permission). If there is any error, `fopen` will return the null pointer value `NULL` (which is defined as zero in `stdio.h`).

The next thing needed is a way to read or write the file once it is open. There are several possibilities, of which `getc` and `putc` are the simplest. `getc` returns the next character from a file; it needs the file pointer to tell it what file. Thus

```
c = getc(fp)
```

places in `c` the next character from the file referred to by `fp`; it returns `EOF` when it reaches end of file. `putc` is the inverse of `getc`:

```
putc(c, fp)
```

puts the character c on the file fp and returns c. `getc` and `putc` return EOF on error.

When a program is started, three files are opened automatically, and file pointers are provided for them. These files are the standard input, the standard output, and the standard error output; the corresponding file pointers are called `stdin`, `stdout`, and `stderr`. Normally these are all connected to the terminal, but may be redirected to files or pipes as described in Section 2.2. `stdin`, `stdout` and `stderr` are pre-defined in the I/O library as the standard input, output and error files; they may be used anywhere an object of type `FILE *` can be. They are constants, however, *not* variables, so don't try to assign to them.

With some of the preliminaries out of the way, we can now write wc. The basic design is one that has been found convenient for many programs: if there are command-line arguments, they are processed in order. If there are no arguments, the standard input is processed. This way the program can be used stand-alone or as part of a larger process.

```
#include <stdio.h>

main(argc, argv)    /* wc: count lines, words, chars */
int argc;
char *argv[];
{
    int c, i, inword;
    FILE *fp, *fopen();
    long linect, wordct, charct;
    long tlinect = 0, twordct = 0, tcharct = 0;

    i = 1;
    fp = stdin;
    do {
        if (argc > 1 && (fp=fopen(argv[i], "r")) == NULL) {
            fprintf(stderr, "wc: can't open %s\n", argv[i]);
            continue;
        }
        linect = wordct = charct = inword = 0;
        while ((c = getc(fp)) != EOF) {
            charct++;
            if (c == '\n')
                linect++;
            if (c == ' ' || c == '\t' || c == '\n')
                inword = 0;
            else if (inword == 0) {
                inword = 1;
                wordct++;
            }
        }
        printf("%7ld %7ld %7ld", linect, wordct, charct);
        printf(argc > 1 ? " %s\n" : "\n", argv[i]);
        fclose(fp);
        tlinect += linect;
        twordct += wordct;
        tcharct += charct;
    } while (++i < argc);
    if (argc > 2)
        printf("%7ld %7ld %7ld total\n", tlinect, twordct, tcharct);
    exit(0);
}
```

The function `fprintf` is identical to `printf`, save that the first argument is a file pointer that specifies the file to be written.

The function `fclose` is the inverse of `fopen`; it breaks the connection between the file pointer and the external name that was established by `fopen`, freeing the file pointer for another file. Since there is a limit on the number of files that a program may have open simultaneously, it's a good idea to free things when they are no longer needed. There is also another reason to call `fclose` on an output file — it flushes the buffer in which `putc` is collecting output. (`fclose` is called automatically for each open file when a program terminates normally.)

3.2. Error Handling — Stderr and Exit

`stderr` is assigned to a program in the same way that `stdin` and `stdout` are. Output written on `stderr` appears on the user's terminal even if the standard output is redirected. `pr` writes its diagnostics on `stderr` instead of `stdout` so that if one of the files can't be accessed for some reason, the message finds its way to the user's terminal instead of disappearing down a pipeline or into an output file.

The program actually signals errors in another way, using the function `exit` to terminate program execution. The argument of `exit` is available to whatever process called it (see Section 6), so the success or failure of the program can be tested by another program that uses this one as a sub-process. By convention, a return value of 0 signals that all is well; non-zero values signal abnormal situations.

`exit` itself calls `fclose` for each open output file, to flush out any buffered output, then calls a routine named `_exit`. The function `_exit` causes immediate termination without any buffer flushing; it may be called directly if desired.

3.3. Miscellaneous I/O Functions

The standard I/O library provides several other I/O functions besides those we have illustrated above.

Normally output with `putc`, etc., is buffered (except to `stderr`); to force it out immediately, use `fflush(fp)`.

`fscanf` is identical to `scanf`, except that its first argument is a file pointer (as with `fprintf`) that specifies the file from which the input comes; it returns `EOF` at end of file.

The functions `sscanf` and `sprintf` are identical to `fscanf` and `fprintf`, except that the first argument names a character string instead of a file pointer. The conversion is done from the string for `sscanf` and into it for `sprintf`.

`fgets(buf, size, fp)` copies the next line from `fp`, up to and including a newline, into `buf`; at most `size-1` characters are copied; it returns `NULL` at end of file. `fputs(buf, fp)` writes the string in `buf` onto file `fp`.

The function `ungetc(c, fp)` "pushes back" the character `c` onto the input stream `fp`; a subsequent call to `getc`, `fscanf`, etc., will encounter `c`. Only one character of pushback per file is permitted.

4. LOW-LEVEL I/O

This section describes the bottom level of I/O on the UNIX system. The lowest level of I/O in UNIX provides no buffering or any other services; it is in fact a direct entry into the operating system. You are entirely on your own, but on the other hand, you have the most control over what happens. And since the calls and usage are quite simple, this isn't as bad as it sounds.

4.1. File Descriptors

In the UNIX operating system, all input and output is done by reading or writing files, because all peripheral devices, even the user's terminal, are files in the file system. This means that a single, homogeneous interface handles all communication between a program and peripheral devices.

In the most general case, before reading or writing a file, it is necessary to inform the system of your intent to do so, a process called "opening" the file. If you are going to write on a file, it may also be necessary to create it. The system checks your right to do so (Does the file exist? Do you have permission to access it?), and if all is well, returns a small positive integer called a *file descriptor*. Whenever I/O is to be done on the file, the file descriptor is used instead of the name to identify the file. (This is roughly analogous to the use of READ(5,...) and WRITE(6,...) in Fortran.) All information about an open file is maintained by the system; the user program refers to the file only by the file descriptor.

The file pointers discussed in section 3 are similar in spirit to file descriptors, but file descriptors are more fundamental. A file pointer is a pointer to a structure that contains, among other things, the file descriptor for the file in question.

Since input and output involving the user's terminal are so common, special arrangements exist to make this convenient. When the command interpreter (the "shell") runs a program, it opens three files, with file descriptors 0, 1, and 2, called the standard input, the standard output, and the standard error output. All of these are normally connected to the terminal, so if a program reads file descriptor 0 and writes file descriptors 1 and 2, it can do terminal I/O without worrying about opening the files.

If I/O is redirected to and from files with < and >, as in

```
prog <infile >outfile
```

the shell changes the default assignments for file descriptors 0 and 1 from the terminal to the named files. Similar observations hold if the input or output is associated with a pipe. Normally file descriptor 2 remains attached to the terminal, so error messages can go there. In all cases, the file assignments are changed by the shell, not by the program. The program does not need to know where its input comes from nor where its output goes, so long as it uses file 0 for input and 1 and 2 for output.

4.2. Read and Write

All input and output is done by two functions called `read` and `write`. For both, the first argument is a file descriptor. The second argument is a buffer in your program where the data is to come from or go to. The third argument is the number of bytes to be transferred. The calls are

```
n_read = read(fd, buf, n);

n_written = write(fd, buf, n);
```

Each call returns a byte count which is the number of bytes actually transferred. On reading, the number of bytes returned may be less than the number asked for, because fewer than `n` bytes remained to be read. (When the file is a terminal, `read` normally reads only up to the next newline, which is generally less than what was requested.) A return value of zero bytes implies end of file, and -1 indicates an error of some sort. For writing, the returned value is the number of bytes actually written; it is generally an error if this isn't equal to the number supposed to be written.

The number of bytes to be read or written is quite arbitrary. The two most common values are 1, which means one character at a time ("unbuffered"), and 512, which corresponds to a physical blocksize on many peripheral devices. This latter size will be most efficient, but even character at a time I/O is not inordinately expensive.

Putting these facts together, we can write a simple program to copy its input to its output. This program will copy anything to anything, since the input and output can be redirected to any file or device.

```
#define   BUFSIZE   512  /* best size for PDP-11 UNIX */

main()    /* copy input to output */
{
     char buf[BUFSIZE];
     int  n;

     while ((n = read(0, buf, BUFSIZE)) > 0)
          write(1, buf, n);
     exit(0);
}
```

If the file size is not a multiple of BUFSIZE, some `read` will return a smaller number of bytes to be written by `write`; the next call to `read` after that will return zero.

It is instructive to see how `read` and `write` can be used to construct higher level routines like `getchar`, `putchar`, etc. For example, here is a version of `getchar` which does unbuffered input.

```
#define   CMASK     0377 /* for making char's > 0 */

getchar() /* unbuffered single character input */
{
     char c;

     return((read(0, &c, 1) > 0) ? c & CMASK   EOF);
}
```

c *must* be declared `char`, because `read` accepts a character pointer. The character being returned must be masked with 0377 to ensure that it is positive; otherwise sign extension may make it negative. (The constant 0377 is appropriate for the PDP-11 but not necessarily for other machines.)

The second version of `getchar` does input in big chunks, and hands out the characters one at a time.

```
#define   CMASK     0377 /* for making char's > 0 */
#define   BUFSIZE   512

getchar() /* buffered version */
{
     static char    buf[BUFSIZE];
     static char    *bufp = buf;
     static int     n = 0;

     if (n == 0) {  /* buffer is empty */
          n = read(0, buf, BUFSIZE);
          bufp = buf;
     }
     return((--n >= 0) ? *bufp++ & CMASK   EOF);
}
```

4.3. Open, Creat, Close, Unlink

Other than the default standard input, output and error files, you must explicitly open files in order to read or write them. There are two system entry points for this, `open` and `creat` [sic].

`open` is rather like the `fopen` discussed in the previous section, except that instead of returning a file pointer, it returns a file descriptor, which is just an `int`.

```
int fd;

fd = open(name, rwmode);
```

As with `fopen`, the `name` argument is a character string corresponding to the external file name. The access mode argument is different, however: `rwmode` is 0 for read, 1 for write, and 2 for read and write access. `open` returns -1 if any error occurs; otherwise it returns a valid file descriptor.

It is an error to try to `open` a file that does not exist. The entry point `creat` is provided to create new files, or to re-write old ones.

```
fd = creat(name, pmode);
```

returns a file descriptor if it was able to create the file called `name`, and -1 if not. If the file already exists, `creat` will truncate it to zero length; it is not an error to `creat` a file that already exists.

If the file is brand new, `creat` creates it with the *protection mode* specified by the `pmode` argument. In the UNIX file system, there are nine bits of protection information associated with a file, controlling read, write and execute permission for the owner of the file, for the owner's group, and for all others. Thus a three-digit octal number is most convenient for specifying the permissions. For example, 0755 specifies read, write and execute permission for the owner, and read and execute permission for the group and everyone else.

To illustrate, here is a simplified version of the UNIX utility *cp*, a program which copies one file to another. (The main simplification is that our version copies only one file, and does not permit the second argument to be a directory.)

```
#define NULL 0
#define BUFSIZE 512
#define PMODE 0644 /* RW for owner, R for group, others */

main(argc, argv)    /* cp: copy f1 to f2 */
int argc;
char *argv[];
{
    int  f1, f2, n;
    char buf[BUFSIZE];

    if (argc != 3)
        error("Usage: cp from to", NULL);
    if ((f1 = open(argv[1], 0)) == -1)
        error("cp: can't open %s", argv[1]);
    if ((f2 = creat(argv[2], PMODE)) == -1)
        error("cp: can't create %s", argv[2]);

    while ((n = read(f1, buf, BUFSIZE)) > 0)
        if (write(f2, buf, n) != n)
            error("cp: write error", NULL);
    exit(0);
}

error(s1, s2)  /* print error message and die */
char *s1, *s2;
{
    printf(s1, s2);
    printf("\n");
    exit(1);
}
```

As we said earlier, there is a limit (typically 15-25) on the number of files which a program may have open simultaneously. Accordingly, any program which intends to process many files must be prepared to re-use file descriptors. The routine `close` breaks the connection between a file descriptor and an open file, and frees the file descriptor for use with some other file. Termination of a program via `exit` or return from the main program closes all open files.

The function `unlink(filename)` removes the file `filename` from the file system.

4.4. Random Access — Seek and Lseek

File I/O is normally sequential: each `read` or `write` takes place at a position in the file right after the previous one. When necessary, however, a file can be read or written in any arbitrary order. The system call `lseek` provides a way to move around in a file without actually reading or writing:

```
lseek(fd, offset, origin);
```

forces the current position in the file whose descriptor is `fd` to move to position `offset`, which is taken relative to the location specified by `origin`. Subsequent reading or writing will begin at that position. `offset` is a `long`; `fd` and `origin` are `int`'s. `origin` can be 0, 1, or 2 to specify that `offset` is to be measured from the beginning, from the current position, or from the end of the file respectively. For example, to append to a file, seek to the end before writing:

```
lseek(fd, 0L, 2);
```

To get back to the beginning ("rewind"),

```
lseek(fd, 0L, 0);
```

Notice the `0L` argument; it could also be written as `(long) 0`.

With `lseek`, it is possible to treat files more or less like large arrays, at the price of slower access. For example, the following simple function reads any number of bytes from any arbitrary place in a file.

```
get(fd, pos, buf, n) /* read n bytes from position pos */
int fd, n;
long pos;
char *buf;
{
     lseek(fd, pos, 0);  /* get to pos */
     return(read(fd, buf, n));
}
```

In pre-version 7 UNIX, the basic entry point to the I/O system is called `seek`. `seek` is identical to `lseek`, except that its `offset` argument is an `int` rather than a `long`. Accordingly, since PDP-11 integers have only 16 bits, the `offset` specified for `seek` is limited to 65,535; for this reason, `origin` values of 3, 4, 5 cause `seek` to multiply the given offset by 512 (the number of bytes in one physical block) and then interpret `origin` as if it were 0, 1, or 2 respectively. Thus to get to an arbitrary place in a large file requires two seeks, first one which selects the block, then one which has `origin` equal to 1 and moves to the desired byte within the block.

4.5. Error Processing

The routines discussed in this section, and in fact all the routines which are direct entries into the system can incur errors. Usually they indicate an error by returning a value of -1. Sometimes it is nice to know what sort of error occurred; for this purpose all these routines, when appropriate, leave an error number in the external cell `errno`. The meanings of the various error numbers are listed in the introduction to Section II of the *UNIX Programmer's Manual*, so your program can, for example, determine if an attempt to open a file failed

because it did not exist or because the user lacked permission to read it. Perhaps more commonly, you may want to print out the reason for failure. The routine `perror` will print a message associated with the value of `errno`; more generally, `sys_errno` is an array of character strings which can be indexed by `errno` and printed by your program.

5. PROCESSES

It is often easier to use a program written by someone else than to invent one's own. This section describes how to execute a program from within another.

5.1. The "System" Function

The easiest way to execute a program from another is to use the standard library routine `system`. `system` takes one argument, a command string exactly as typed at the terminal (except for the newline at the end) and executes it. For instance, to time-stamp the output of a program,

```
main()
{
        system("date");
        /* rest of processing */
}
```

If the command string has to be built from pieces, the in-memory formatting capabilities of `sprintf` may be useful.

Remember than `getc` and `putc` normally buffer their input; terminal I/O will not be properly synchronized unless this buffering is defeated. For output, use `fflush`; for input, see `setbuf` in the appendix.

5.2. Low-Level Process Creation — Execl and Execv

If you're not using the standard library, or if you need finer control over what happens, you will have to construct calls to other programs using the more primitive routines that the standard library's `system` routine is based on.

The most basic operation is to execute another program *without returning*, by using the routine `execl`. To print the date as the last action of a running program, use

```
execl("/bin/date", "date", NULL);
```

The first argument to `execl` is the *file name* of the command; you have to know where it is found in the file system. The second argument is conventionally the program name (that is, the last component of the file name), but this is seldom used except as a place-holder. If the command takes arguments, they are strung out after this; the end of the list is marked by a `NULL` argument.

The `execl` call overlays the existing program with the new one, runs that, then exits. There is *no* return to the original program.

More realistically, a program might fall into two or more phases that communicate only through temporary files. Here it is natural to make the second pass simply an `execl` call from the first.

The one exception to the rule that the original program never gets control back occurs when there is an error, for example if the file can't be found or is not executable. If you don't know where `date` is located, say

```
execl("/bin/date", "date", NULL);
execl("/usr/bin/date", "date", NULL);
fprintf(stderr, "Someone stole 'date'\n");
```

A variant of `execl` called `execv` is useful when you don't know in advance how many arguments there are going to be. The call is

```
execv(filename, argp);
```

where `argp` is an array of pointers to the arguments; the last pointer in the array must be `NULL` so `execv` can tell where the list ends. As with `execl`, `filename` is the file in which the program is found, and `argp[0]` is the name of the program. (This arrangement is identical to the `argv` array for program arguments.)

Neither of these routines provides the niceties of normal command execution. There is no automatic search of multiple directories — you have to know precisely where the command is located. Nor do you get the expansion of metacharacters like <, >, *, ?, and [] in the argument list. If you want these, use `execl` to invoke the shell `sh`, which then does all the work. Construct a string `commandline` that contains the complete command as it would have been typed at the terminal, then say

```
execl("/bin/sh", "sh", "-c", commandline, NULL);
```

The shell is assumed to be at a fixed place, `/bin/sh`. Its argument -c says to treat the next argument as a whole command line, so it does just what you want. The only problem is in constructing the right information in `commandline`.

5.3. Control of Processes — Fork and Wait

So far what we've talked about isn't really all that useful by itself. Now we will show how to regain control after running a program with `execl` or `execv`. Since these routines simply overlay the new program on the old one, to save the old one requires that it first be split into two copies; one of these can be overlaid, while the other waits for the new, overlaying program to finish. The splitting is done by a routine called `fork`:

```
proc_id = fork();
```

splits the program into two copies, both of which continue to run. The only difference between the two is the value of `proc_id`, the "process id." In one of these processes (the "child"), `proc_id` is zero. In the other (the "parent"), `proc_id` is non-zero; it is the process number of the child. Thus the basic way to call, and return from, another program is

```
if (fork() == 0)
    execl("/bin/sh", "sh", "-c", cmd, NULL);      /* in child */
```

And in fact, except for handling errors, this is sufficient. The `fork` makes two copies of the program. In the child, the value returned by `fork` is zero, so it calls `execl` which does the `command` and then dies. In the parent, `fork` returns non-zero so it skips the `execl`. (If there is any error, `fork` returns -1).

More often, the parent wants to wait for the child to terminate before continuing itself. This can be done with the function `wait`:

```
int status;

if (fork() == 0)
    execl(...);
wait(&status);
```

This still doesn't handle any abnormal conditions, such as a failure of the `execl` or `fork`, or the possibility that there might be more than one child running simultaneously. (The `wait` returns the process id of the terminated child, if you want to check it against the value returned by `fork`.) Finally, this fragment doesn't deal with any funny behavior on the part of the child (which is reported in `status`). Still, these three lines are the heart of the standard library's `system` routine, which we'll show in a moment.

The `status` returned by `wait` encodes in its low-order eight bits the system's idea of the child's termination status; it is 0 for normal termination and non-zero to indicate various kinds of problems. The next higher eight bits are taken from the argument of the call to `exit` which caused a normal termination of the child process. It is good coding practice for all programs to

return meaningful status.

When a program is called by the shell, the three file descriptors 0, 1, and 2 are set up pointing at the right files, and all other possible file descriptors are available for use. When this program calls another one, correct etiquette suggests making sure the same conditions hold. Neither `fork` nor the `exec` calls affects open files in any way. If the parent is buffering output that must come out before output from the child, the parent must flush its buffers before the `execl`. Conversely, if a caller buffers an input stream, the called program will lose any information that has been read by the caller.

5.4. Pipes

A *pipe* is an I/O channel intended for use between two cooperating processes: one process writes into the pipe, while the other reads. The system looks after buffering the data and synchronizing the two processes. Most pipes are created by the shell, as in

```
ls | pr
```

which connects the standard output of `ls` to the standard input of `pr`. Sometimes, however, it is most convenient for a process to set up its own plumbing; in this section, we will illustrate how the pipe connection is established and used.

The system call `pipe` creates a pipe. Since a pipe is used for both reading and writing, two file descriptors are returned; the actual usage is like this:

```
int fd[2];

stat = pipe(fd);
if (stat == -1)
    /* there was an error */
```

`fd` is an array of two file descriptors, where `fd[0]` is the read side of the pipe and `fd[1]` is for writing. These may be used in `read`, `write` and `close` calls just like any other file descriptors.

If a process reads a pipe which is empty, it will wait until data arrives; if a process writes into a pipe which is too full, it will wait until the pipe empties somewhat. If the write side of the pipe is closed, a subsequent `read` will encounter end of file.

To illustrate the use of pipes in a realistic setting, let us write a function called `popen(cmd, mode)`, which creates a process `cmd` (just as `system` does), and returns a file descriptor that will either read or write that process, according to `mode`. That is, the call

```
fout = popen("pr", WRITE);
```

creates a process that executes the `pr` command; subsequent `write` calls using the file descriptor `fout` will send their data to that process through the pipe.

`popen` first creates the the pipe with a `pipe` system call; it then `fork`s to create two copies of itself. The child decides whether it is supposed to read or write, closes the other side of the pipe, then calls the shell (via `execl`) to run the desired process. The parent likewise closes the end of the pipe it does not use. These closes are necessary to make end-of-file tests work properly. For example, if a child that intends to read fails to close the write end of the pipe, it will never see the end of the pipe file, just because there is one writer potentially active.

```
#include <stdio.h>

#define   READ 0
#define   WRITE     1
#define   tst(a, b) (mode == READ ? (b)    (a))
static    int  popen_pid;

popen(cmd, mode)
char *cmd;
int  mode;
{
     int p[2];

     if (pipe(p) < 0)
          return(NULL);
     if ((popen_pid = fork()) == 0) {
          close(tst(p[WRITE], p[READ]));
          close(tst(0, 1));
          dup(tst(p[READ], p[WRITE]));
          close(tst(p[READ], p[WRITE]));
          execl("/bin/sh", "sh", "-c", cmd, 0);
          _exit(1); /* disaster has occurred if we get here */
     }
     if (popen_pid == -1)
          return(NULL);
     close(tst(p[READ], p[WRITE]));
     return(tst(p[WRITE], p[READ]));
}
```

The sequence of `close`s in the child is a bit tricky. Suppose that the task is to create a child process that will read data from the parent. Then the first `close` closes the write side of the pipe, leaving the read side open. The lines

```
close(tst(0, 1));
dup(tst(p[READ], p[WRITE]));
```

are the conventional way to associate the pipe descriptor with the standard input of the child. The `close` closes file descriptor 0, that is, the standard input. `dup` is a system call that returns a duplicate of an already open file descriptor. File descriptors are assigned in increasing order and the first available one is returned, so the effect of the `dup` is to copy the file descriptor for the pipe (read side) to file descriptor 0; thus the read side of the pipe becomes the standard input. (Yes, this is a bit tricky, but it's a standard idiom.) Finally, the old read side of the pipe is closed.

A similar sequence of operations takes place when the child process is supposed to write from the parent instead of reading. You may find it a useful exercise to step through that case.

The job is not quite done, for we still need a function `pclose` to close the pipe created by `popen`. The main reason for using a separate function rather than `close` is that it is desirable to wait for the termination of the child process. First, the return value from `pclose` indicates whether the process succeeded. Equally important when a process creates several children is that only a bounded number of unwaited-for children can exist, even if some of them have terminated; performing the `wait` lays the child to rest. Thus:

```
#include <signal.h>

pclose(fd)      /* close pipe fd */
int fd;
{
     register r, (*hstat)(), (*istat)(), (*qstat)();
     int   status;
     extern int popen_pid;

     close(fd);
     istat = signal(SIGINT, SIG_IGN);
     qstat = signal(SIGQUIT, SIG_IGN);
     hstat = signal(SIGHUP, SIG_IGN);
     while ((r = wait(&status)) != popen_pid && r != -1);
     if (r == -1)
          status = -1;
     signal(SIGINT, istat);
     signal(SIGQUIT, qstat);
     signal(SIGHUP, hstat);
     return(status);
}
```

The calls to `signal` make sure that no interrupts, etc., interfere with the waiting process; this is the topic of the next section.

The routine as written has the limitation that only one pipe may be open at once, because of the single shared variable `popen_pid`; it really should be an array indexed by file descriptor. A `popen` function, with slightly different arguments and return value is available as part of the standard I/O library discussed below. As currently written, it shares the same limitation.

6. SIGNALS — INTERRUPTS AND ALL THAT

This section is concerned with how to deal gracefully with signals from the outside world (like interrupts), and with program faults. Since there's nothing very useful that can be done from within C about program faults, which arise mainly from illegal memory references or from execution of peculiar instructions, we'll discuss only the outside-world signals: *interrupt*, which is sent when the DEL character is typed; *quit*, generated by the FS character; *hangup*, caused by hanging up the phone; and *terminate*, generated by the *kill* command. When one of these events occurs, the signal is sent to *all* processes which were started from the corresponding terminal; unless other arrangements have been made, the signal terminates the process. In the *quit* case, a core image file is written for debugging purposes.

The routine which alters the default action is called `signal`. It has two arguments: the first specifies the signal, and the second specifies how to treat it. The first argument is just a number code, but the second is the address is either a function, or a somewhat strange code that requests that the signal either be ignored, or that it be given the default action. The include file `signal.h` gives names for the various arguments, and should always be included when signals are used. Thus

```
#include <signal.h>
 ...
signal(SIGINT, SIG_IGN);
```

causes interrupts to be ignored, while

```
signal(SIGINT, SIG_DFL);
```

restores the default action of process termination. In all cases, `signal` returns the previous value of the signal. The second argument to `signal` may instead be the name of a function (which has to be declared explicitly if the compiler hasn't seen it already). In this case, the named routine will be called when the signal occurs. Most commonly this facility is used to

allow the program to clean up unfinished business before terminating, for example to delete a temporary file:

```
#include <signal.h>

main()
{
     int onintr();

     if (signal(SIGINT, SIG_IGN) != SIG_IGN)
          signal(SIGINT, onintr);

     /* Process      */

     exit(0);
}

onintr()
{
     unlink(tempfile);
     exit(1);
}
```

Why the test and the double call to `signal`? Recall that signals like interrupt are sent to *all* processes started from a particular terminal. Accordingly, when a program is to be run non-interactively (started by `&`), the shell turns off interrupts for it so it won't be stopped by interrupts intended for foreground processes. If this program began by announcing that all interrupts were to be sent to the `onintr` routine regardless, that would undo the shell's effort to protect it when run in the background.

The solution, shown above, is to test the state of interrupt handling, and to continue to ignore interrupts if they are already being ignored. The code as written depends on the fact that `signal` returns the previous state of a particular signal. If signals were already being ignored, the process should continue to ignore them; otherwise, they should be caught.

A more sophisticated program may wish to intercept an interrupt and interpret it as a request to stop what it is doing and return to its own command-processing loop. Think of a text editor: interrupting a long printout should not cause it to terminate and lose the work already done. The outline of the code for this case is probably best written like this:

```
#include <signal.h>
#include <setjmp.h>
jmp_buf   sjbuf;

main()
{
     int (*istat)(), onintr();

     istat = signal(SIGINT, SIG_IGN);   /* save original status */
     setjmp(sjbuf); /* save current stack position */
     if (istat != SIG_IGN)
          signal(SIGINT, onintr);

     /* main processing loop */
}
```

```
onintr()
{
     printf("\nInterrupt\n");
     longjmp(sjbuf);      /* return to saved state */
}
```

The include file `setjmp.h` declares the type `jmp_buf` an object in which the state can be saved. `sjbuf` is such an object; it is an array of some sort. The `setjmp` routine then saves the state of things. When an interrupt occurs, a call is forced to the `onintr` routine, which can print a message, set flags, or whatever. `longjmp` takes as argument an object stored into by `setjmp`, and restores control to the location after the call to `setjmp`, so control (and the stack level) will pop back to the place in the main routine where the signal is set up and the main loop entered. Notice, by the way, that the signal gets set again after an interrupt occurs. This is necessary; most signals are automatically reset to their default action when they occur.

Some programs that want to detect signals simply can't be stopped at an arbitrary point, for example in the middle of updating a linked list. If the routine called on occurrence of a signal sets a flag and then returns instead of calling `exit` or `longjmp`, execution will continue at the exact point it was interrupted. The interrupt flag can then be tested later.

There is one difficulty associated with this approach. Suppose the program is reading the terminal when the interrupt is sent. The specified routine is duly called; it sets its flag and returns. If it were really true, as we said above, that "execution resumes at the exact point it was interrupted," the program would continue reading the terminal until the user typed another line. This behavior might well be confusing, since the user might not know that the program is reading; he presumably would prefer to have the signal take effect instantly. The method chosen to resolve this difficulty is to terminate the terminal read when execution resumes after the signal, returning an error code which indicates what happened.

Thus programs which catch and resume execution after signals should be prepared for "errors" which are caused by interrupted system calls. (The ones to watch out for are reads from a terminal, `wait`, and `pause`.) A program whose `onintr` program just sets `intflag`, resets the interrupt signal, and returns, should usually include code like the following when it reads the standard input:

```
if (getchar() == EOF)
     if (intflag)
          /* EOF caused by interrupt */
     else
          /* true end-of-file */
```

A final subtlety to keep in mind becomes important when signal-catching is combined with execution of other programs. Suppose a program catches interrupts, and also includes a method (like "!" in the editor) whereby other programs can be executed. Then the code should look something like this:

```
if (fork() == 0)
     execl(...);
signal(SIGINT, SIG_IGN); /* ignore interrupts */
wait(&status); /* until the child is done */
signal(SIGINT, onintr);  /* restore interrupts */
```

Why is this? Again, it's not obvious but not really difficult. Suppose the program you call catches its own interrupts. If you interrupt the subprogram, it will get the signal and return to its main loop, and probably read your terminal. But the calling program will also pop out of its wait for the subprogram and read your terminal. Having two processes reading your terminal is very unfortunate, since the system figuratively flips a coin to decide who should get each line of input. A simple way out is to have the parent program ignore interrupts until the child is done. This reasoning is reflected in the standard I/O library function `system`:

```
#include <signal.h>

system(s) /* run command string s */
char *s;
{
     int status, pid, w;
     register int (*istat)(), (*qstat)();

     if ((pid = fork()) == 0) {
          execl("/bin/sh", "sh", "-c", s, 0);
          _exit(127);
     }
     istat = signal(SIGINT, SIG_IGN);
     qstat = signal(SIGQUIT, SIG_IGN);
     while ((w = wait(&status)) != pid && w != -1)
          ;
     if (w == -1)
          status = -1;
     signal(SIGINT, istat);
     signal(SIGQUIT, qstat);
     return(status);
}
```

As an aside on declarations, the function `signal` obviously has a rather strange second argument. It is in fact a pointer to a function delivering an integer, and this is also the type of the signal routine itself. The two values `SIG_IGN` and `SIG_DFL` have the right type, but are chosen so they coincide with no possible actual functions. For the enthusiast, here is how they are defined for the PDP-11; the definitions should be sufficiently ugly and nonportable to encourage use of the include file.

```
#define   SIG_DFL   (int (*)())0
#define   SIG_IGN   (int (*)())1
```

References

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

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

[3] B. W. Kernighan, "UNIX for Beginners — Second Edition." Bell Laboratories, 1978.

Appendix — The Standard I/O Library

D. M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

The standard I/O library was designed with the following goals in mind.

1. It must be as efficient as possible, both in time and in space, so that there will be no hesitation in using it no matter how critical the application.
2. It must be simple to use, and also free of the magic numbers and mysterious calls whose use mars the understandability and portability of many programs using older packages.
3. The interface provided should be applicable on all machines, whether or not the programs which implement it are directly portable to other systems, or to machines other than the PDP-11 running a version of UNIX.

1. General Usage

Each program using the library must have the line

```
#include <stdio.h>
```

which defines certain macros and variables. The routines are in the normal C library, so no special library argument is needed for loading. All names in the include file intended only for internal use begin with an underscore _ to reduce the possibility of collision with a user name. The names intended to be visible outside the package are

`stdin` The name of the standard input file

`stdout` The name of the standard output file

`stderr` The name of the standard error file

`EOF` is actually −1, and is the value returned by the read routines on end-of-file or error.

`NULL` is a notation for the null pointer, returned by pointer-valued functions to indicate an error

`FILE` expands to `struct _iob` and is a useful shorthand when declaring pointers to streams.

`BUFSIZ` is a number (viz. 512) of the size suitable for an I/O buffer supplied by the user. See `setbuf`, below.

`getc, getchar, putc, putchar, feof, ferror, fileno`
are defined as macros. Their actions are described below; they are mentioned here to point out that it is not possible to redeclare them and that they are not actually functions; thus, for example, they may not have breakpoints set on them.

The routines in this package offer the convenience of automatic buffer allocation and output flushing where appropriate. The names `stdin`, `stdout`, and `stderr` are in effect constants and may not be assigned to.

2. Calls

`FILE *fopen(filename, type) char *filename, *type;`
opens the file and, if needed, allocates a buffer for it. `filename` is a character string specifying the name. `type` is a character string (not a single character). It may be `"r"`, `"w"`, or `"a"` to indicate intent to read, write, or append. The value returned is a file pointer. If it is `NULL` the attempt to open failed.

`FILE *freopen(filename, type, ioptr) char *filename, *type; FILE *ioptr;`

The stream named by `ioptr` is closed, if necessary, and then reopened as if by `fopen`. If the attempt to open fails, `NULL` is returned, otherwise `ioptr`, which will now refer to the new file. Often the reopened stream is `stdin` or `stdout`.

`int getc(ioptr) FILE *ioptr;`
returns the next character from the stream named by `ioptr`, which is a pointer to a file such as returned by `fopen`, or the name `stdin`. The integer `EOF` is returned on end-of-file or when an error occurs. The null character `\0` is a legal character.

`int fgetc(ioptr) FILE *ioptr;`
acts like `getc` but is a genuine function, not a macro, so it can be pointed to, passed as an argument, etc.

`putc(c, ioptr) FILE *ioptr;`
`putc` writes the character `c` on the output stream named by `ioptr`, which is a value returned from `fopen` or perhaps `stdout` or `stderr`. The character is returned as value, but `EOF` is returned on error.

`fputc(c, ioptr) FILE *ioptr;`
acts like `putc` but is a genuine function, not a macro.

`fclose(ioptr) FILE *ioptr;`
The file corresponding to `ioptr` is closed after any buffers are emptied. A buffer allocated by the I/O system is freed. `fclose` is automatic on normal termination of the program.

`fflush(ioptr) FILE *ioptr;`
Any buffered information on the (output) stream named by `ioptr` is written out. Output files are normally buffered if and only if they are not directed to the terminal; however, `stderr` always starts off unbuffered and remains so unless `setbuf` is used, or unless it is reopened.

`exit(errcode);`
terminates the process and returns its argument as status to the parent. This is a special version of the routine which calls `fflush` for each output file. To terminate without flushing, use `_exit`.

`feof(ioptr) FILE *ioptr;`
returns non-zero when end-of-file has occurred on the specified input stream.

`ferror(ioptr) FILE *ioptr;`
returns non-zero when an error has occurred while reading or writing the named stream. The error indication lasts until the file has been closed.

`getchar();`
is identical to `getc(stdin)`.

`putchar(c);`
is identical to `putc(c, stdout)`.

`char *fgets(s, n, ioptr) char *s; FILE *ioptr;`
reads up to `n-1` characters from the stream `ioptr` into the character pointer `s`. The read terminates with a newline character. The newline character is placed in the buffer followed by a null character. `fgets` returns the first argument, or `NULL` if error or end-of-file occurred.

`fputs(s, ioptr) char *s; FILE *ioptr;`
writes the null-terminated string (character array) `s` on the stream `ioptr`. No newline is appended. No value is returned.

`ungetc(c, ioptr) FILE *ioptr;`

The argument character c is pushed back on the input stream named by ioptr. Only one character may be pushed back.

```
printf(format, a1, ...) char *format;
fprintf(ioptr, format, a1, ...) FILE *ioptr; char *format;
sprintf(s, format, a1, ...)char *s, *format;
```

printf writes on the standard output. fprintf writes on the named output stream. sprintf puts characters in the character array (string) named by s. The specifications are as described in section printf(3) of the *UNIX Programmer's Manual*.

```
scanf(format, a1, ...) char *format;
fscanf(ioptr, format, a1, ...) FILE *ioptr; char *format;
sscanf(s, format, a1, ...) char *s, *format;
```

scanf reads from the standard input. fscanf reads from the named input stream. sscanf reads from the character string supplied as s. scanf reads characters, interprets them according to a format, and stores the results in its arguments. Each routine expects as arguments a control string format, and a set of arguments, *each of which must be a pointer*, indicating where the converted input should be stored.

scanf returns as its value the number of successfully matched and assigned input items. This can be used to decide how many input items were found. On end of file, EOF is returned; note that this is different from 0, which means that the next input character does not match what was called for in the control string.

```
fread(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr;
```
reads nitems of data beginning at ptr from file ioptr. No advance notification that binary I/O is being done is required; when, for portability reasons, it becomes required, it will be done by adding an additional character to the mode-string on the fopen call.

```
fwrite(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr;
```
Like fread, but in the other direction.

```
rewind(ioptr) FILE *ioptr;
```
rewinds the stream named by ioptr. It is not very useful except on input, since a rewound output file is still open only for output.

```
system(string) char *string;
```
The string is executed by the shell as if typed at the terminal.

```
getw(ioptr) FILE *ioptr;
```
returns the next word from the input stream named by ioptr. EOF is returned on end-of-file or error, but since this a perfectly good integer feof and ferror should be used. A "word" is 16 bits on the PDP-11.

```
putw(w, ioptr) FILE *ioptr;
```
writes the integer w on the named output stream.

```
setbuf(ioptr, buf) FILE *ioptr; char *buf;
```
setbuf may be used after a stream has been opened but before I/O has started. If buf is NULL, the stream will be unbuffered. Otherwise the buffer supplied will be used. It must be a character array of sufficient size:

```
char buf[BUFSIZ];
```

```
fileno(ioptr) FILE *ioptr;.
```
returns the integer file descriptor associated with the file.

```
fseek(ioptr, offset, ptrname) FILE *ioptr; long offset;
```
The location of the next byte in the stream named by ioptr is adjusted. offset is a long integer. If ptrname is 0, the offset is measured from the beginning of the file; if ptrname is 1, the offset is measured from the current read or write pointer; if ptrname is 2, the offset is measured from the end of the file. The routine accounts properly for any buffering. (When

this routine is used on non-UNIX systems, the offset must be a value returned from `ftell` and the ptrname must be 0).

`long ftell(ioptr) FILE *ioptr;`
The byte offset, measured from the beginning of the file, associated with the named stream is returned. Any buffering is properly accounted for. (On non-UNIX systems the value of this call is useful only for handing to `fseek`, so as to position the file to the same place it was when `ftell` was called.)

`getpw(uid, buf) char *buf;`
The password file is searched for the given integer user ID. If an appropriate line is found, it is copied into the character array `buf`, and 0 is returned. If no line is found corresponding to the user ID then 1 is returned.

`char *malloc(num);`
allocates `num` bytes. The pointer returned is sufficiently well aligned to be usable for any purpose. `NULL` is returned if no space is available.

`char *calloc(num, size);`
allocates space for `num` items each of size `size`. The space is guaranteed to be set to 0 and the pointer is sufficiently well aligned to be usable for any purpose. `NULL` is returned if no space is available .

`cfree(ptr) char *ptr;`
Space is returned to the pool used by `calloc`. Disorder can be expected if the pointer was not obtained from `calloc`.

The following are macros whose definitions may be obtained by including `<ctype.h>`.

`isalpha(c)` returns non-zero if the argument is alphabetic.

`isupper(c)` returns non-zero if the argument is upper-case alphabetic.

`islower(c)` returns non-zero if the argument is lower-case alphabetic.

`isdigit(c)` returns non-zero if the argument is a digit.

`isspace(c)` returns non-zero if the argument is a spacing character: tab, newline, carriage return, vertical tab, form feed, space.

`ispunct(c)` returns non-zero if the argument is any punctuation character, i.e., not a space, letter, digit or control character.

`isalnum(c)` returns non-zero if the argument is a letter or a digit.

`isprint(c)` returns non-zero if the argument is printable — a letter, digit, or punctuation character.

`iscntrl(c)` returns non-zero if the argument is a control character.

`isascii(c)` returns non-zero if the argument is an ascii character, i.e., less than octal 0200.

`toupper(c)` returns the upper-case character corresponding to the lower-case letter `c`.

`tolower(c)` returns the lower-case character corresponding to the upper-case letter `c`.

A68 Assembler

Reference Manual

This document describes the syntax and usage of the **a68** assembler for the Motorola 68000 microprocessor. The basic format of **a68** is loosely based on the Digital Equipment Corp Macro-11 assembler described in DEC's publication DEC-11-OMACA-A-D, but also contains elements of the UNIX *as* assembler. The instruction mnemonics and effective address format are derived from a Motorola publication on the 68000, *the MACSS MC68000 Design Specification Instruction Set Processor* dated June30, 1979.

Sections 1-3 of this document describe the general form of **a68** programs, section 4 describes the instruction mnemonics and addressing modes, section 5 describes the pseudo-ops supported by the assembler and section 6 describes the error codes generated. For instructions on how to operate the assembler from UNIX, readers should type in *man a68*.

May, 1981

Massachusetts Institute of Technology
Laboratory of Computer Science
Cambridge, Massachusetts

Author's initials: DK

Acknowledgement: To those unknown guys wherever they are who implemented the **a68** compiler belong our thanks. Denis Bzowy did the adaption for the MUNIX system and added the listing option. Dittmar Krall massaged this user's guide a little and did some few error correction.

TABLE OF CONTENTS

1. NOTATION

The notation used in this document is a modified BNF similar to that used in the MULTICS PL/I Language Manual. The operators of the BNF in order of decreasing precedence are:

x ...	Repetition: Denotes one or more occurrences of **x**.
xy	Juxtaposition: Denotes an occurrence of **x** followed by an occurrence of **y**
x|y	Alternation: Denotes an occurrence of **x** or **y** but not both.

Brackets and braces define the order of expression interpretation. The subexpression enclosed in brackets is optional. That is,

[x]	denotes zero or one occurrence of x.
{x|y}z	denotes an x or a y, followed by a z.

Brackets or braces which appear in a68 syntax will be boldfaced, to distinguish them from the meta- brackets and braces.

2. SOURCE PROGRAM FORMAT

An a68 program consists of a series of statements, each of which occupies exactly one line, i.e., a sequence of characters followed by the *newline* character. Form feed, ascii ^L, also serves as a line terminator. Neither multiple statements on a single line nor continuation lines are allowed.

The format of an a68 assembly language statement is:

[LabelField :] op-code [OperandField] [|comment]

There are three exceptions to this rule:

1. Blank lines are permitted.
2. A statement may contain only a *LabelField*. The label defined in this field has the same value as if it were defined in the *LabelField* of the next statement in the program. For example, the two statements

   ```
   sea:
   movw d1,d2
   ```

   are equivalent to the single statement

   ```
   sea: movw d1,d2
   ```

3. A line may consist of only the comment field. For example, the two statements below are allowed:

```
| This is a comment field.
| So is this
```

In general, blanks or tabs are allowed anywhere in a statement. For example, multiple blanks are allowed in the *OperandField* to separate symbols from operators. Blanks are meaningful only when they occur in a character string (e.g. as the operand of an *.ascii* pseudo-op). At least one blank or tab must appear between the op-code and the *OperandField* of a statement.

2.1. Label Fields

A label is a user-defined symbol which is assigned the value of the current location counter and entered into the assembler's symbol table. The value of the label may be either absolute or relocatable; in the latter case, the absolute value of the symbol is assigned when the program is linked via *ld*.

A label is a symbolic means of referring to a specific location within a program. If present, a label **always** occurs first in a statement and **must** be terminated by a colon. The collection of label definitions in a statement is called the *LabelField*.

The format of a *LabelField* is:

symbol: [symbol:] . . .

Examples:

```
start:
sea: bar:     | Multiple symbols
7$:           | A local symbol, defined below
```

2.2. Op-code Fields

The *OpcodeField* of an assembly language statement identifies the statement as either a machine instruction, or an assembler directive. One or more blanks (or tabs) must separate the *OpcodeField* from the *OperandField* in a statement. No blanks are necessary between *LabelFields* and *OpcodeFields*, but they are recommended to improve readability of the program.

A machine instruction is indicated by an instruction mnemonic. The assembly language statement is intended to produce a single executable machine instruction. The operation of each instruction is described in the manufacturer's user manual. Some conventions used in **a68** for instruction mnemonics are described in section 4 and a complete list of the instructions is presented in the appendix.

An assembler directive, or pseudo-op, performs some function during the assembly process. It does not produce any executable code, but it may assign space in a program for data.

2.3. Operand Fields

A distinction is made between *OperandField* and *operand* in **a68**. Several machine instructions and assembler directives require two or more arguments, and each of these is referred to as an *operand*. In general, an *OperandField* consists of zero or more operands, and in all cases, operands are separated by a comma. In other words, the format for an *OperandField* is:

[operand [, operand] . . .]

The format of the *OperandField* for machine instruction statements is the same for all instructions, and is described in section 4. The format of the *OperandField* for assembler directives depends on the directive itself, and is included in the directive's description in section 5 of this manual.

2.4. Comment Field

The comment character in **a68** is the vertical bar, (|), not the semicolon, (;). Use of the semicolon as a comment character will result in an "Invalid Operator" error.

The comment field consists of all characters on a source line following and including the comment character. These characters are ignored by the assembler. Any character may appear in the comment field, with the obvious exception of the *newline* character, which starts a new line.

A |- line switches listing off. A |+ line switches listing on.

3. SYMBOLS AND EXPRESSIONS

This section describes the various components of **a68** expressions: symbols, numbers, terms, and expressions.

3.1. Symbols

A symbol consists of a sequence of characters, with the following restrictions:

1. Valid characters include A-Z, a-z, 0-9, period (.), underscore (_), and dollar sign ($).
2. The first character must not be numeric.

All characters are significant and are checked in comparisons with other symbols. Upper and lower cases are distinct, ("One" and "one" are separate symbols).

A symbol is declared when the assembler recognizes it as a symbol of the program. A symbol is defined when a value is associated with it. With the exception of symbols declared by a *.globl* directive, all symbols are defined when they are declared. A label symbol (which represents an address of the program) may not be redefined; all other symbols are allowed to receive a new value.

There are several ways to declare a symbol:

1. As the label of a statement (See section 2.1).
2. In a *direct assignment* statement.
3. As an *external* symbol via the *.globl* directive.
4. As a *common* symbol via the *.comm* directive.
5. As a *local* symbol.

3.2. Direct Assignment Statements

A *direct assignment* statement assigns the value of an arbitrary expression to a specified symbol. The format of a *direct assignment* statement is:

symbol = expression

Examples of valid *direct assignments* are:

```
vect_size = 4
vectora = 0xFFFE
vectorb = vectora - vect_size
CRLF = 0x0D0A
```

Only one symbol may be assigned in a single statement.

Any symbol defined by *direct assignment* may be redefined later in the program, in which case its value is the result of the last such statement. A *local* symbol may be defined by *direct assignment*, though this doesn't make much sense. Label or register symbols may not be redefined.

If the *expression* is absolute, then the symbol is also absolute, and may be treated as a constant in subsequent expressions (see section 3.4). If the *expression* is relocatable, however, then the symbol is also relocatable, and it is considered to be declared in the same program section as the *expression*. See section 3.7 for an explanation of absolute and relocatable expressions.

If the *expression* contains an *external* symbol, then the symbol defined by the = statement will also be considered *external*. For example:

```
.globl x | x is declared as external symbol
sum = x  | sum becomes an external symbol
```

assigns the value of x (zero if it is undefined) to sum and makes sum an *external* symbol. *External* symbols may be defined by *direct assignment*.

3.3. Register Symbols

Register symbols are symbols used to represent registers in the machine. Register symbols are defined in the source descriptor file for a machine in the pre-assembly code portion of the file. This portion consists of source statements that are assembled before every source program for the machine.

The following symbols are register symbols.

d0	d1	d2	d3	d4	d5	d6	d7
a0	a1	a2	a3	a4	a5	a6	a7
sp	pc	cc	sr	usp			

3.4. External Symbols

A program may be assembled in separate modules, and then linked together to form a single program (see *ld* (I)). *External* symbols may be defined in each of these separate modules. A symbol which is declared

(given a value) in one module may be referenced in another module by declaring the symbol to be *external* in **both** modules. There are two forms of *external* symbols: those defined with the *.globl* and those defined with the *.comm* directive.

External symbols are declared with the *.globl* assembler directive. The format is:

.globl symbol [, symbol] . . .

For example, the following statements declare the array TABLE and the routine SRCH to be *external* symbols:

```
.globl TABLE,SRCH
TABLE: .=.+20
SRCH: movl #TABLE,a0
```

External symbols are only **declared** to the assembler. *External* symbols must be **defined** (i.e. given a value) in some other statement by one of the methods mentioned above. They need not be defined in the current program; in this case they are flagged as "undefined" in the symbol table. If they are undefined, they are considered to have a value of zero in expressions.

The other form of *external* symbol is defined with the *.comm* directive. These statement reserve storage in the bss section similar to FORTRAN common areas. The format of the statement is:

.comm name, ConstantExpression

which causes **a68** to declare the *name* as a *common* symbol with a value equal to the *ConstantExpression*. For the rest of the assembly this symbol will be treated as though it was an undefined global. **a68** does not allocate storage for *common* symbols; this task is left to the loader. The loader will compute the maximum size of for each *common* symbol which may appear in several load modules, allocates storage for it in the final *bss* section and resolves linkages.

3.5. Local Symbols

Local symbols provide a convenient means of generating labels for branch instructions, etc. Use of *local* symbols reduces the possibility of multiply-defined symbols in a program, and separates entry point symbols from local references, such as the top of a loop. Local symbols cannot be referenced by other object modules.

Local symbols are of the form **n$** where **n** is any integer. The following are valid *local* symbols

1$
27$
3943$

A *local* symbol is defined and referenced only within a single "local symbol block" (lsb). A new local symbol block is entered when:

1. a label is declared; or,
2. a new program section is entered.

There is no conflict between *local* symbols with the same name which appear in different local symbol blocks.

3.6. Assembly Location Counter

The assembly location counter is the period character, '.'; hence its name "dot". When used in the operand field of any statement, dot represents the address of the first byte of the statement. Even in assembly directives, it represents the address of the start of the directive. A dot appearing as the third argument in a *.byte* instruction has the value of the address where the first byte was loaded; this address is not updated "during" the pseudo-op.

For example,

```
Ralph: movl .,a0  |load value of this instruction into a0
```

At the beginning of each assembly pass, the assembler clears the location counter. Normally, consecutive memory locations are assigned to each byte of generate code. However, the location where the code is stored may be changed by a *direct assignment* altering the location counter:

```
. = expression
```

This *expression* must not contain any forward references, and must not change from one pass to another. Storage area may also be reserved by advancing dot. For example, if the current value of dot is 1000, the *direct assignment* statement:

```
Table: .=.+100
```

would reserve 100 (decimal) bytes of storage, with the address of the first byte as the value of Table. The next instruction would be stored at address 1100.

3.7. Program Sections

As in UNIX, programs to **a68** are divided into three sections: *text*, *data* and *bss*. The normal interpretation of these sections is: instruction space, initialized data space and uninitialized data space, respectively. These three sections are equivalent as far as **a68** is concerned with the exception that no instructions or data will be output for the *bss* section although its size will be computed and its symbol values will be output.

In the first pass of the assembly, **a68** maintains a separate location counter for each section, thus for code like:

```
.text
sum: movw d1,d2
.data
hello: .long 27
.text
jnk: addw d2,d1
.data
myst: .byte 4
```

in the output, sum will immediately precede jnk and hello will immediately precede myst. At the end of the first pass, **a68** rearranges all the addresses so that the sections will be output in the following order: *text*, *data* and *bss*. The resulting output file is an executable image file with all addresses correctly resolved, with the exception of undefined *.globls* and *.comms*. For more information on the format of the output file, consult the UNIX *man* entry on *a.out* files.

3.8. Constants

All constants are considered absolute quantities when appearing in an expression.

3.8.1. Numeric Constants

An **a68** numeric constant is a sequence of digits. **a68** interprets integers as octal, hex, or decimal according to the following conventions.

octal	octal numbers begin with 0
hex	hex numbers begin with 0x or 0X
decimal	all other numbers

3.9. Operators

3.9.1. Unary Operators

There are two unary operators in **a68**:

-	unary minus.
~	logical negation.

3.9.2. Binary Operators

Binary operators in **a68** include:

+	Addition; e.g. "3+4" evaluates to 7.
-	Subtraction; e.g. "3-4" evaluates to -1., or 0xFFFFFFFF
*	Multiplication; e.g. "4*3" evaluates to 12.

Each operator is assumed to work on a 32-bit number.

3.10. Terms

A term is a component of an expression. A term may be one of the following:

1. A number.
2. A symbol.
3. A term preceded by a unary operator. For example, both "sum" and "~sum" may be considered to be terms. Multiple unary operators are allowed; e.g. " -- A" has the same value as "A".

3.11. Expressions

Expressions are combinations of terms joined together by binary operators. An expression is always evaluated to a 32-bit value. If the instruction calls for only one byte, (e.g. *.byte*), then the low-order byte is used.

Expressions are evaluated left to right with no operator precedence. Thus "1+2*3" evaluates to 9, not 7. Unary operators have precedence over binary operators since they are considered part of a term, and both terms of a binary operator must be evaluated before the binary operator can be applied.

A missing expression or term is interpeted as having a value of zero. In this case, an "Invalid expression" error will be generated. An "Invalid Operator" error means that a valid end-of-line character or binary operator was not detected after the assembler processed a term. In particular, this error will be generated if an expression contains a symbol with an illegal character, or if an incorrect comment character was used.

Any expression, when evaluated, is either absolute, relocatable, or external:

a. An expression is absolute if its value is fixed. An expression whose terms are constants, or symbols whose values are constants via a *direct assignment* statement, is absolute. A relocatable expression minus a relocatable term, where both items belong to the same program section is also absolute.

b. An expression is relocatable if its value is fixed relative to a base address, but will have an offset value when it is linked, or loaded into core. All labels of a program defined in relocatable sections are relocatable terms, and any expression which contains them must only add or subtract constants to their value. For example, assume the symbol "sum" was defined in a relocatable section of the program. Then the following demonstrates the use of relocatable expressions:

sum	relocatable
sum+5	relocatable
sum*2	Not relocatable (error)
2-sum	Not relocatable (error), since the expression cannot be linked by adding sum's offset to it.
sum-jnk	Absolute, since the offsets added to sum and jnk cancel each other out.

c. An expression is *external* (or global) if it contains an *external* symbol not defined in the current program. The same restrictions on expressions containing relocatable symbols apply to expressions containing *external* symbols. Exception: the expression sum-jnk where both sum and jnk are *external* symbols is not allowed.

4. INSTRUCTIONS AND ADDRESSING MODES

This section describes the conventions used in **a68** to specify instruction mnemonics and addressing modes.

4.1. Instruction Mnemonics

The instruction mnemonics used by **a68** are described in the previously mentioned Motorola manual with a few variations. Most of the 68000 instructions can apply to byte, word on long operands, so in **a68** the normal instruction mnemonic is suffixed with **b**, **w**, or **l** to indicate which length operand was intended. For example, there are three mnemonics for the *or* instruction: *orb*, *orw* and *orl*. Op-codes for instructions with unusual opcodes may have additional suffixes, thus in addition to the normal add variations, there also exist: *addqb*, *addqw* and *addql* for the add quick instruction.

Branch instructions come in two flavors, byte and word. In **a68**, the byte (i.e., short) version is specified by appending the suffix **s** to the basic mnemonic as in *beq* and *beqs*.

In addition to the instructions which explicitly specify the instruction length, **a68** supports extended branch instructions, whose names are generally constructed by replacing the **b** with **j**. If the operand of the extended branch instruction is a simple address in the current segment, and the offset to that address is sufficiently small, **a68** will automatically generate the corresponding short branch instruction. If the offset is too large for a short branch, but small enough for a branch, then the corresponding branch instruction is generated. If the operand references an external address or is complex, then the extended branch instruction is implemented either by a *jmp* or *jsr* (for *jra* or *jbsr*), or by a conditional branch (with the sense of the conditional inverted) around a *jmp* for the extended conditional branches. In this context, a complex address is either an address which specifies other than normal mode addressing, or relocatable expressions containing more than one relocatable symbol. i.e. if **a**, **b** and **c** are symbols in the current segment, then the expression **a+b-c** is relocatable, but not simple.

Note that **a68** is not optimal for extended branch instructions whose operand addresses the next instruction. The optimal code is no instruction at all, but **a68** currently retains insufficient information to make this optimization. The difficulty is that if **a68** decides to just eliminate the instruction, the address of the next instruction will be the same as the address of the (nonexistent) extended branch instruction. This instruction will then look like a branch to the current location, which would require an instruction to be generated. The code that **a68** actually generates for this case is a nop (recall that an offset of zero in a branch instruction indicates a long offset). Although this problem may arise in compiler code generators, it can easily be handled by a peephole

optimizer.

The algorithm used by **a68** for deciding how to implement extended branch instructions is described in "Assembling Code for Machines with Span-Dependent Instruction," by Thomas G. Szymanski in Communications of the ACM, Volume 21, Number 4, pp300-308, April 1978.

Consult the appendix for a complete list of the instruction op-codes.

4.2. Addressing Modes

The following table describes the addressing modes recognized by **a68**. In this table **an** refers to an address register, **dn** refers to a data register, **Ri** to either a data or an address register, **d** to a displacement, which, in **a68** is a constant expression, and **xxx** to a constant expression. Certain instructions, particularly *move* accept a variety of special registers including **sp**, the stack pointer which is equivalent to **a7**, **sr** †, the status register, **cc**, the condition codes of the status register, **usp**, the user mode stack pointer, and **pc**, the program counter.

Mode	Notation	Example
Register	an,dn,sp,pc,cc,sr,usp	movw a3,d2
Register Deferred	an@	movw a3@,d2
Postincrement	an@+	movw a3@+,d2
Predecrement	an@-	movw a3@-,d2
Displacement	an@(d)	movw a3@(24),d2
Word Index	an@(d, Ri:W)	movw a3@(16, d2:W),d3
Long Index	an@(d, Ri:L)	movw a3@(16, d2:L),d3
Absolute Short	xxx:W ‡	movw 14:W,d2
Absolute Long	xxx:L	movw 14:L,d2
PC Displacement	pc@(d)	movw pc@(20),d3
PC Word Index	pc@(d, Ri:W)	movw pc@(14, d2:W),d3
PC Long Index	pc@(d, Ri:L)	movw pc@(14, d2:L),d3
Normal	sun	movw sun,d3
Immediate	#xxx	movw #27+3,d3

Normal mode actually assembles as absolute long, although the value of the constant will be flagged as relocatable to the loader. The notation for these modes derived from the Motorola notation with the exception of the colon in index mode rather than period.

The Motorola manual presents different opcodes for instructions that use the effective address as data rather than the contents of the effective address such as *adda* for add address. **a68** does not make this distinction because it can determine the type of the operand from its form. Thus an

† The access to **sr** is restricted to specific 68000 instructions.

‡ MUNIX LD(1) does not support short external addresses

instruction of the form:

```
sun: .word 0
     ...
     addl #sun,a0
```

will assemble to the *add address* instruction because **sun** is known to be an address.

The 68000 tends to be very restrictive in that most instructions accept only a limited subset of the address modes above. For example, the add address instruction does not accept a data register as a destination. **a68** tries to check all these restrictions and will generate the illegal operand error code for instructions that do not satisfy the address mode restrictions.

5. ASSEMBLER DIRECTIVES

The following pseudo-ops are available in **a68**:

.ascii	stores character strings
.asciz	"
.byte	stores 8-bit bytes
.word	stores 16-bit words
.long	stores 32-bit longwords
.zerol	long zeroes
.text	Text csect
.data	Data csect
.bss	Bss csect
.globl	declares external symbols
.extern	same as **.globl**
.comm	declares common symbols

5.1. .ascii .asciz

The *.ascii* directive translates character strings into their 7-bit ascii (represented as 8-bit bytes) equivalents for use in the source program. The format of the ascii directive is as follows:

.ascii "character string"

The syntax of C-strings is used (\n is newline, etc.). Obviously, a *newline* must not appear within the character string.

The *.asciz* directive is equivalent to the *.ascii* directive with a zero byte automatically inserted as the final character of the string. Thus, when a list or text string is to be printed, a search for the null character can terminate the string.

5.2. .byte .word .long

The *.byte*, *.word* and *.long* directives are used to reserve bytes and words, and to initialize them with certain values.

The format is:

[label:] .byte [expression] [,expression] .

[label:] .word [expression] [,expression] . .

[label:] .long [expression] [,expression] . .

For example, the first statement reserves one byte for each expression in the operand field, and initializes the value of the byte to be the low-order byte of the corresponding expression. Note that multiple expressions must be separated by commas. A blank expression is interpreted as zero, and no error is generated.

The syntax and semantics for *.word* is identical, except that 16-bit words are reserved and initialized, of course, and *.long* uses 32-bit quantities.

5.3. .text .data .bss

These statements change the "program section" where assembled code will be loaded.

5.4. .globl .comm

See section 4.4.

5.5. .even

This directive advances the location counter if its current value is odd. This is useful for forcing storage allocation like *.word* directives to be on word boundaries.

6. ERROR CODES

If an error is detected during assembly, a message of the form:

line_no .error_code

is output to the standard error stream.

The following *.error_codes*, and their probable cause, appear below:

2 Invalid Character. An invalid character for a character constant or character string was encountered.

3 Multiply defined symbol. A symbol appears twice as a label, or an attempt to redefine a label using an = statement.

4 Symbol storage exceeded. No more room is left in the symbol table. Assemble portions of the program separately, then bind them together.

6 Symbol length exceeded. A symbol of more than 31 characters was encountered.

7 Undefined symbol. A symbol not declared by one of the methods mentioned above in 'Symbols' was encountered. This happens when an invalid instruction mnemonic is used. This also occurs when an invalid or non-printing character occurs in the statement.

8 Invalid Constant. An invalid digit was encountered in a number.

9 Invalid Term. The expression evaluator could not find a valid term: symbol, constant or expression. An invalid prefix to a number or a bad symbol name in an operand will generate this.

10 Invalid Operator. Check the operand field for a bad operator.

11 Non-relocatable expression. If an expression contains a relocatable symbol (e.g. label) then the only operations that can be applied to it are the addition of absolute expressions or the subtraction of another relocatable symbol (which produces an absolute result).

12 Invalid operand type. The type field of an operand is either not defined for the machine, or represents an addressing mode incompatible with the current instruction.

13 Invalid operand. This is a catch-all *.error*. It appears notably when an attempt is made to assign an undefined value to dot during pass 1.

14 Invalid symbol. If the first token on the source line is not a valid symbol (or the beginning of a comment), this is generated. Might happen if you try and implied *.word*.

15 Invalid assignment. An attempt was made to redefine a label with an = statement.

16 Too many labels. More than 10 labels and/or symbol= 's appeared on a single statement.

17 Invalid op-code. An op-code mnemonic was not recognized by the assembler.

18 Invalid entry point. The entry point of the program (declared on the *.end* statement) must be a defined label.

19 Invalid string. An invalid string for *.ascii* or *.asciz* was encountered. Make sure string is enclosed in double quotes.

20 Bad filename or too many levels. An invalid filename was given to *.insrt*, or there were more than 10 levels of nested *.insrts*.

22 *.error* statement. An *.error* statement was encountered during pass 2.

25 Wrong number of operands. This is usually a warning. Check the manufacturer's assembly manual for the correct number of operands for the current instruction.

26 Line too long. A statement with more that 132 characters before the *newline* was encountered.

27 Invalid register expression. Any expression inside parentheses should be absolute and have the value of a register code (register symbols). This may occur if you use parentheses for anything other than the register portion of an operand.

7. Appendix

OpCode	Description
abcd	add decimal with extend
addL	add
addqL	add quick
addxL	add extended
andL	and
aslL	arithmetic shift left
asrL	arithmetic shift right
bCCS	branch on condition
bchg	test a bit and change
bclr	test a bit and clear
bra	branch
bset	test a bit and set
bsrS	subroutine branch
btst	test a bit
chk	check register against bounds
clrL	clear an operand
cmpL	compare
cmpmL	compare memory
divs	signed divide
divu	unsigned divide
eorL	exclusive or
exg	exchange registers
extl	sign extend
extw	sign extend
jbsr	jump to subroutine
jCC	jump on condition
jra	jump
jsr	jump to subroutine
lea	load effective address
link	link
lslL	logical shift left
lsrL	logical shift right
movL	move
moveml	move multiple registers
movemw	move multiple registers

OpCode	Description
movepl	move peripheral
movepw	move peripheral
moveq	move quick
muls	signed multiply
mulu	unsigned multiply
nbcd	negate decimal with extend
negL	negate binary
negxL	negate binary with extend
nop	no operation
notL	logical compliment
orL	inclusive or
pea	push effective address
reset	reset machine
rolL	rotate left
rorL	rotate right
roxlL	rotate left with extend
roxrL	rotate right with extend
rte	return from exception
rtr	return and restore codes
rts	return from subroutine
sbcd	subtract decimal with extend
sCC	set on condition
sf	set all zeros
st	set all ones
stop	halt machine
subL	subtract
subqL	subtract quick
subxL	subtract extended
swap	swap register halves
tas	test operand then set
trap	trap
trapv	trap on overflow
tstL	test operand
unlk	unlink

Where:

S Short branch	
s	short
	long branch

CC Condition Code	
cs	carry set
eq	equal
ra	inconditional
ge	greater or equal
gt	greater than
hi	high
le	less or equal
ls	lower or same
lt	less than
mi	minus
ne	not equal
pl	positive
vc	no overflow
vs	overflow set

L Length	
b	byte
w	word
l	long

This piece of software will not longer be supported

Assembler 68000 User's Guide
Version 1.1
July 1982

ABSTRACT

This user's guide is intended for use in writing and translating assembly programs on 68000 UNIX† systems. The assembler is upward compatible with the M68000 Cross Macro assembler provided by Motorola. This guide restricts itself therefore to describing the extensions provided by us.

The Assembler 68000 is a modified version of the CERN Cross Macro Assembler M68MIL implemented in Pascal. Changes were made to write a.out format (the object module format coming with UNIX) instead of CUFOM (CERN Universal Format for Object Modules).

†UNIX is a Trademark of Bell Laboratories.

CONTENTS

1. INTRODUCTION

The Cross Macro Assembler described in this manual is derived from the CERN Cross Macro Assembler M68MIL but writes a.out object format instead of CUFOM. Some of the pseudo instructions were changed to provide the user with a means to control a.out segments instead of CUFOM sections. All other pseudo instructions, all machine instructions and the expression rules - as far as symbol types are not concerned - are the same as for M68MIL and Motorola's Cross Assembler.

The assembler translates assembler source programs for the Motorola 68000 microprocessor into a.out, the format of UNIX loadable modules. A linkage editor subsequently allows the combination and linking of several such modules into a new a.out module. An archiver (see *ar(1)*) permits the construction of a.out libraries, which can be placed in the input to the linkage editor. Besides, the assembler will accept source files 'piped' through the UNIX preprocessor *cpp*.

The assembler is upward compatible with the M68000 Cross Macro Assembler provided by Motorola. Additional pseudo instructions are provided to allow the generation of relocatable object modules. The user is advised to see the following Motorola publications:

M68000 Cross Macro Assembler Reference Manual
B68KXASM(D3), Third Edition, September 1979

MC68000 16-Bit Microprocessor, User's Manual
MC68000UM(AD2), Second Edition, January, 1980

and the UNIX documentation

UNIX Programmer's Manual, Volume 1 (especially *ar(1)*, *ld(1)*, *a.out(5)*)

as a base for the use of this assembler. This manual will restrict itself to the description of the extensions made to the definitions of the Motorola cross assembler. For the readers' convenience this will be done in complete chapters rather than by listing the explicit differences.

The main areas covered by this note are:

Short description of a.out
Expressions (generalized)
Pseudo instructions
Macro definitions
How to use the assembler

Acknowledgement: I would like to thank Mr. Horst von Eicken who gave us the Pascal sources and user manual of his CERN Cross Assembler. If you are interested in this assembler, please contact him at:

Horst von Eicken
Data Handling Division
CERN
CH 1211 Geneva 23
Switzerland

2. SHORT DESCRIPTION OF a.out

In a.out modules code and data fall into three segments: the text segment, the data segment and the bss segment. The text segment is the one in which the assembler begins, and it is the one into which instructions are typically placed. The UNIX system can enforce the purity of the text segment of programs by trapping write operations into it (see *ld(1)*).

The data segment is available for placing data or instructions which will be modified during execution. Anything which may go in the text segment may be put into the data segment. In programs with write-protected, shareable text segments, data segment contains the initialized but variable parts of a program.

The bss segment may not contain any explicitely initialized code or data. The length of the bss segment (like that of text or data) is determined by the high-water mark of the location counter within it. At the start of execution of a program, the bss segment is set up by statements such as:

```
lab  dsl 2
```

Another a.out convention concerns the entry point to a program: a program starts at a label __entry (which is typically defined by some runtime system and does some basic initialization) and branches then to a user-defined label _main. The user has to make sure that one and only one _main label is defined within her/his program.

☛ Since a.out modules are always relocatable it is not possible in an a.out module to allocate a label (symbol) at a certain absolute address - as it can be reached elsewhere by means of an org pseudo instruction - rather than by arranging the link-edit input appropriately (see *ld(1)*). org is supported for compatibility but it's limited.

3. SYMBOLS AND EXPRESSIONS

3.1. Symbols

Symbols recognized by the assembler consist of one or more characters, the first sixteen of which are significant. The first character must be a letter (a through z and A through Z) or an underscore (_), each remaining character may be a letter, a digit (0 through 9) or an underscore. The names for registers (a0 through a7, d0 through d7,sp,usp,ccr,sr), instructions (abcd .. unlnk) and pseudo instructions (blong .. ttl) are predefined symbols and may not be redefined by the user.

☛ a and A are different; predefined names must be written in lower case.

Numbers recognized by the assembler include decimal, hexadecimal and octal values. Decimal numbers are specified by a string of decimal digits (0 through 9); hexadecimal numbers are specified by a dollar sign ($), followed by a string of hexadecimal digits (0 through 9, A through F, a through f); octal numbers are specified by a colon (:), followed by a string of octal digits (0 through 7).

One or more characters enclosed by apostrophes (') constitute a character string. Character strings are left-adjusted and zero-filled (if necessary), whether stored or used as immediate operands. Only strings of four or fewer characters may be used as immediate operands. (In order to specify an apostrophe within a character string, two successive apostrophes must appear where the single apostrophe is intended to appear.)

The assembler has six types of symbols:

absolute symbol:

1. The symbol is equated (equ) or set to an absolute value.
2. The symbol is equated (equ) or set to a constant. Its value is unaffected by any possible future applications of the link- editor to the module.

text symbol:

1. The symbol is equated (equ) or set to a text symbol.
2. The symbol is defined in the text segment of the program. Its value is measured with respect to the beginning of the text segment of the program. If the assembler output is link-edited, its text symbols may change in value since the module need not be the first in the link editor's output. At the start of an assembly the value of "*" is text 0.

data symbol:

1. The symbol is equated (equ) or set to a data symbol.
2. The symbol is defined in the data segment of the program. Its value is measured with respect to the beginning of the data segment of the program. If the assembler output is link-edited, its data symbols may change in value since the module need not be the first in the link editor's output. After the first data pseudo instruction the value of "*" is data 0.

bss symbol:

1. The symbol is equated (equ) or set to a bss:symbol.
2. The symbol is defined in the bss segment of the program. Its value is measured with respect to the beginning of the bss segment of the program. Like text and data symbols, the value of a bss symbol may change during a subsequent link-editor run, since previously loaded programs may have bss segments. After the first bss pseudo instruction the value of "*" is bss 0.

external symbol:

The symbol is listed in a extern pseudo instruction and is not defined in the current assembly. Its value is set to zero and must be defined during a subsequent link-editor run.

undefined symbol:

The symbol is neither defined in the current assembly nor listed in an extern pseudo instruction. The occurrence of such a symbol is indicated as an error.

All symbols except absolute symbols are relative, i.e. they represent relocatable addresses. Whenever the assembler encounters a text, data, or bss symbol it will generate a direct address (see below) and related relocation information. To yield a program counter relative address write

<rsym>(pc) or
<rsym>(pc,<reg>)

respectively.

Symbols defined within an assembly as absolute, text, data, or bss symbol may be exported by occurring in a entry pseudo instruction, i.e. their value and their type are available to the link-editor so that the program may be loaded with others that reference these symbols.

3.2. Expressions

An expression is a combination of symbols, constants, algebraic operators, and parentheses. The expression is used to specify a value which is to be used as an operand. Expressions follow the conventional rules of algebra.

Expressions may contain relative symbols. However the following rules must be followed in order for the expression to be valid:

1. Relative symbols or expressions cannot be multiplied, divided, added, or operated on with the logical operators.
2. A relative symbol or expression may have an absolute value added to or subtracted from it. The result is relative.
3. A relative symbol or expression may be subtracted from another relative symbol or expression provided they are both defined and of same type. The result is absolute.

3.3. Direct Addressing

(The addressing mode discussed here is called 'absolute addressing mode' in the related Motorola documentation. Since those address values cannot be changed at run time, but may be changed during a link-edit run, they are referred to here as direct addresses to avoid confusion with absolute symbols

defined ealier in this manual.)

Since the M 68000 microprocessor allows two forms of direct addressing,

- short direct address (16 bit address) -
- long direct address (32 bit address) -

the assembler has to take a decision as to which form to assume whenever it encounters a forward referenced absolute symbol, i.e. a symbol which has not yet been defined, or a relative symbol, i.e. a symbol the value of which may be changed by the link-editor. By default it will use the long direct address to ensure correct handling of the symbol. The pseudo instructions **fshort**, **flong**, **text**, **data** and **bss** will allow the programmer to override the assembler defaults. Assembling external symbols it will always use the long direct address mode.

A similar problem exists for branches. If a forward reference is found in a branch instruction, the assembler will use the two-word form of the instruction. Using the suffix .s for the branch instruction the programmer can force the assembler to generate the one-word form of the branch instruction. The pseudo instructions **bshort** and **blong** allow additional control.

4. PSEUDO INSTRUCTIONS

Pseudo Instructions discussed in this chapter are classified according to application as follows:

Module identification (**ident, end**)
Segment control (**text, data, bss**)
Symbol definition (**equ, set**)
Module linkage (**entry, extern**)
Data generation and storage reservation (**dc, ds, org**)
Conditional assembly (**else, endc, endif, ifeq, ifne**)
Source stream control (**insert**)
Listing control (**list,g,nolist,page,nopage,sttl,ttl,fail**)
Object code control (**blong, bshort, flong, fshort, noobj**)
Cross reference control (**xrefon, xrefoff**)

The next chapter describes the definition and use of macros. The format description for the pseudo instructions uses symbols which have the following syntactical meaning:

{..} Enclose optional fields of the pseudo instruction.

<.> Enclose a 'syntactic variable', e.g. <number>.

4.1. Module Identification

4.1.1. IDENT - Module Identification

For compatibility with M68MIL an ident pseudo instruction is accepted but is ignored. It has the following format:

```
ident    <symbol>
```

<symbol>
The symbol defines a name (originally that of the module).

4.1.2. END - End of Module

An end pseudo instruction must be the last instruction of each module. It causes the assembler to terminate all counters, conditional assembly, or macro generation. It also causes the a.out module to be terminated.

```
end {<symbol>}
```

<symbol>
An optional symbol is accepted - for compatibility - but ignored. The program main entrypoint is determined by the label _main as stated in the a.out description.

4.2. Segment Control

Segment control pseudo instructions allow the programmer to divide a source module into separately controlled regions of a program, providing her/him with a means of changing type and value of the location counter. They start or resume assembly for one of the three a.out segments. The segment in use is the segment into which code is subsequently assembled. By default the assembler will always start with the text segment using long direct address mode (see 3.3). If the suffix .s is appended to a segment control pseudo instruction, the assembler will assume that this segment will finally be loaded into low address

memory and use direct short addresses for backward references.

☛ The suffix .s does not imply that forward references in that segment will also be resolved with direct short addresses. fshort must be used for this purpose.

4.2.1. Text Segment

text{.s}

Description:
The pseudo instruction causes the assembler to start or resume assembly in the text segment.

4.2.2. Data Segment

data{.s}

Description:
The pseudo instruction causes the assembler to start or resume assembly in the data segment.

4.2.3. Bss Segment

bss{.s}

Description:
The pseudo instruction causes the assembler to start or resume storage allocation in the bss segment (uninitialized data: ds and org only).

4.3. Symbol Definition

4.3.1. EQU - Equate Symbol Value.

<symbol> equ <expression>

<symbol>
A location symbol following the naming rules must be defined.

<expression>
An expression following the expression rules. Forward references are not allowed.

Description:
An equ pseudo instruction permanently defines the symbol in the location field as having the value and type indicated by the expression in the variable field.

4.3.2. SET - Set or reset symbol value.

<symbol> set <expression>

<symbol>
A location symbol following the naming rules must be defined.

<expression>
An expression following the expression rules. Forward references are not allowed.

Description:
A **set** pseudo instruction defines the symbol in the location field as having the value and type indicated by the expression in the variable field. A subsequent **set** using the same symbol redefines the symbol to the new value and type.

4.4. Module Linkage

The pseudo instructions **entry** and **extern** do not define symbols but either declare symbols defined within a module as being available outside the module or declare symbols referred to in the module as being defined outside the module.

4.4.1. ENTRY - Declare Entry Symbols

entry <sym$_1$>, <sym$_2$>,..., <sym$_n$>

<sym$_i$>
Linkage symbol. Each symbol must be defined in the module as nonexternal (must not be listed on an **extern** pseudo instruction).

Description:
The **entry** pseudo instruction specifies which of the symbolic addresses defined in the module can be referred to by modules assembled independently; **entry** lists entry points to the current module.

4.4.2. EXTERN - Declare External Symbols

extern <sym$_1$>, <sym$_2$>,..., <sym$_n$>

<sym$_i$>
Linkage symbol. These symbols must not be defined within the module.

Description:
The **extern** pseudo instruction lists symbols that are defined as entry points in independently assembled modules for which references can appear in the module being assembled.

4.5. Data Generation and Storage Reservation

4.5.1. DC - Define Constant

{<symbol>} **dc** <opr$_1$>, <opr$_2$>,..., <opr$_n$>
{<symbol>} **dc.b** <opr$_1$>, <opr$_2$>,..., <opr$_n$>
{<symbol>} **dc.w** <opr$_1$>, <opr$_2$>,..., <opr$_n$>
{<symbol>} **dc.l** <opr$_1$>, <opr$_2$>,..., <opr$_n$>

<symbol>
A symbol following the naming rules may be defined.

<opr_1>
The operand can be a symbol, an ascii, decimal or hexadecimal value or an expression evaluating to such a value.

Description:
The function of the dc pseudo instruction is to define a constant in memory. The dc directive may have one operand, or multiple operands which are separated by commas. The operand field may contain a value (decimal, octal, hexadecimal, or character string), a symbol or a expression. The constant is aligned on a word boundary if word(.w) or long (.l) is specified, or a byte boundary if byte (.b) is specified. The constant is limited to 60 bytes.

The following rules apply to size specifications on character strings:

dc.b If an odd number of bytes (characters) are entered, the odd byte on the right will be zero filled unless the next source statement is another dc.b or ds.b. In this case the next dc.b or ds.b will start in the odd byte on the right.

dc.w If an odd number of bytes (characters) are entered, the last word will be zero filled on the right to force an even byte count.

dc.l If less than a multiple of four bytes are entered, the last long word will be zero filled on the right to a multiple of four bytes.

4.5.2. DS - Define Storage

```
{<symbol>}   ds   <expression>
{<symbol>}   ds.b <expression>
{<symbol>}   ds.w <expression>
{<symbol>}   ds.l <expression>
```

<symbol>
A symbol following the naming rules may be defined.

<expression>
The expression must evaluate to an absolute, positive value.

Description:
The ds pseudo instruction is used to reserve memory locations. The contents of the memory reserved are zero filled for text and data segment, for bss segment they are not initialized in any way. The expression must evaluate to an absolute value. Forward references are not allowed.

4.5.3. ORG - Origin

Since a.out modules are always relocatable the org pseudo instruction is applicable only in a very restricted manner: it can be used to advance the location counter a certain number of bytes or to a certain label. The skipped locations are zero filled.

```
org <expression>
```

Description:

The **org** pseudo instruction is used to advance the location counter <expression> bytes. It's identical with

ds.b <expression>*

i.e. the expression must be of current segment type and evaluate to an value greater than the actual location counter.

4.8. Conditional Assembly

The pseudo instructions **ifeq** and **ifne** permit optional assembly or skipping of source code. The instructions immediately following the test instruction are assembled if the tested condition is true and skipped if the condition is false. Skipping is terminated either by a source statement count on the **if** instruction, or by an **endif**, **else** or an **end**. The statement count, when used, is decremented for instruction lines only; comment lines (identified by * in column one) are not counted.

The result of an **if** test is determined by the value of the expression in pass one of the assembler; the value of a relative symbol is relative to the origin of the segment in which it was defined. The value of an external symbol is zero if the symbol was declared as external. If the symbol was defined relative to a declared external, the value is the relative value. **if**'s may be nested up to ten levels deep.

4.8.1. ENDIF - End of IF Range

{<if_name>} **endif**

<if_name>

An optional symbol; defines the name of an **ifeq**, **ifne** or **else** sequence; or blank.

Description:

An **endif** pseudo instruction (or **endc** for compatibility with the Motorola assembler) causes termination of skipping and assembly to resume. When the sequence containing the **endif** is being assembled, or is controlled by a statement count, the **endif** has no effect other than to be included in the count.

Skipped instructions such as macro references are not expanded. Thus, any **endif** that would have resulted from an expansion is not detected.

Skipping of a sequence initiated by an **ifeq**, **ifne** or **else** that is assigned a name is terminated by an **endif** specifying the same name. Skipping of a sequence initiated by an unnamed **ifeq**, **ifne** or **else** is terminated by an unnamed **endif**.

4.6.2. ELSE - Reverse Effects of IF.

{<if_name>} else

<if_name>
An optional symbol; defines the name of an ifeq, ifne or else sequence; or blank.

Description:
By means of the else instruction, the assembler provides the facility to reverse the effects of an if test within the if range. An else detected during skipping causes assembly to resume at the instruction following the else. An else detected while a sequence is being assembled initiates skipping of source code following the else. Skipping continues until either an end or an endif for the sequence is detected.

An else specifying the sequence by name terminates skipping of a sequence initiated by an ifeq or ifne with the same name. An unnamed else terminates skipping of a sequence initiated by an unnamed ifeq or ifne. Skipped instructions such as macro references are not expanded; any else that would have resulted from the expansion is not detected.

4.6.3. IFEQ - Test Expression is Equal Zero.

4.6.4. IFNE - Test Expression is Not Equal Zero.

{<if_name>} ifeq <expression>{,<line_count>}
{<if_name>} ifne <expression>{,<line_count>}

<if_name>
An optional symbol, defines the name of the ifeq or ifne sequence; or blank.

<expression>
A simple expression without forward reference. If the expression is erroneous, an error message is printed and assembly continues with the next instruction.

<line_count>
Optional absolute value specifying an integer count of the number of statements to be skipped.

Description:
The ifeq and ifne pseudo instructions test the value of the expression and assemble instruction in the if range when the condition is satisfied.

The <line_count>, if specified, takes precedence over an <if_name>, if specified at all.

4.7. Source Stream Control

4.7.1. INSERT - Insert Secondary Source.

insert

Description:

The **insert** pseudo instruction provides a means of obtaining source statements from a file other than that being used for input. The assembler transfers the text from this file and assembles it before taking the next statement from the interrupted source of statements.

There are no parameters for the **insert** pseudo instruction. The file to be used is specified when the assembler is called (see 6.) The file will be rewound before using it. Under UNIX, the preprocessor *cpp* supports the inclusion of files and some other features. *cpp* may be used with the assembler (see 6.). Its usage is described in

Kernighan B. W.; Ritchie D. M.:
The C Programming Language.
Prentice-Hall Software Series; Chapter 4.11

4.8. Listing Control

The pseudo instructions described in this section permit extensive control of the assembly listing format.

4.8.1. LIST - Listing

list <op$_1$>, <op$_2$>,..., <op$_n$>

<op$_i$>

Optional parameter. A list option or a list option prefixed by a minus sign. The unprefixed option selects the option; the prefixed option cancels it. Options are separated by commas and terminated by a blank. The following options are available:

dc When dc is selected, the source line of the dc pseudo instruction and its expansion are listed, otherwise only the source line will be listed.
-dc is the default.

if When if is selected, the source lines of the **ifeq**, **ifne**, **else** or **endif** pseudo instructions and the skipped source statements in the if range are listed, otherwise the pseudo instructions are listed, but not the skipped source statements.
-if is the default.

macro
When macro is selected, the source line of the macro reference and the fully expanded macro body are listed, otherwise only the source line of the outermost macro reference of possibly nested macro calls is listed.
-macro is the default.

xopc
When xopc is selected, the assembler will list the use of all opcodes in the cross reference list.
-xopc is the default.

xpse
When xpse is selected, the assembler will list the use of all pseudo instruction in the cross reference list.
-xpse is the default.

xreg
When xreg is selected, the assembler will list the use of all registers in the cross reference list.
-xreg is the default.

Description:
The list pseudo instruction controls the content and format of the assembler listing. Use of the list pseudo instruction is optional. If not specified in a module, or if specified without parameters, the assembler will produce an output according to the default for each possible option.

For compatibility with the Motorola assembler the pseudo instruction g is also accepted.

```
g
```

Description:.
The effect of the g pseudo instruction is identical with the effect of

```
list dc
```

4.8.2. NOLIST - Turn off Listing

```
nolist
nol
```

Description:
The nolist pseudo instruction suppresses the printing of the assembly listing until a list pseudo instruction is encountered.

4.8.3. PAGE - Top of Page.

```
page
```

Description:
The page pseudo instruction advances printer paper to a new page before printing. Then page headings are printed and listing continues. The page pseudo instruction does not appear on the program listing.

4.8.4. NOPAGE - Turn off Paging.

```
nopage
```

Description:
The nopage pseudo instruction suppresses paging to the output device. Page and line numbers in the cross reference and error listing will be meaningless.

4.8.5. SPC - Space Between Source Lines.

```
spc <count>
```

<count>
An absolute value.

Description:
The spc pseudo instruction causes the assembler to output <count> blank lines to the assembly listing. The spc pseudo instruction does not appear on the program listing.

4.8.6. TTL - Assembly Listing Title.

4.8.7. STTL - Assembly Listing Subtitle.

```
ttl  '<text>'
sttl '<text>'
```

Description:
The ttl and sttl pseudo instruction allow the programmer to print a title and a subtitle on the top of each page of the listing. To this effect the assembler maintains internally two text strings which are set to blank at the beginning of pass one. In pass two, whenever a new page is started, these two text strings together with other information are printed in the page header. Specifying a title or subtitle merely means, that the contents of the corresponding internal text string is changed to the one specified with the ttl or sttl pseudo instruction. It does not imply an automatic start of a new page. The first specified title is in addition kept in a third internal text string and is copied into the title text string at the start of pass two. Neither the ttl nor the sttl pseudo instruction are listed in the assembly listing.

4.8.8. FAIL - Generate an Error Message.

```
fail
```

Description:
An error message is printed on the assembly listing.

4.9. Object Code Control

The pseudo instructions blong, bshort, flong or fshort allow the programmer to influence the assembler's choice whenever forward references or relative symbols are encountered, be it for direct addresses or relative branching instructions.

4.9.1. BLONG - Use Two-Word Branch.

4.9.2. BSHORT - Use One-Word Branch.

blong
bshort

Description:

The two pseudo instructions **blong** and **bshort** allow the programmer to influence the assembler whenever it is assembling a forward reference. By default the assembler will use the two-word instruction form allowing a larger relative address range. After a **bshort** pseudo instruction the assembler will generate the one-word relative branch instruction, unless the suffix .l a **blong** pseudo instruction forces the two-word relative branch instruction to be generated, unless the suffix .s has been appended to that branch instruction.

4.9.3. FLONG - Force Direct Long Address.

4.9.4. FSHORT - Force Direct Short Address.

flong
fshort

Description:

The two pseudo instructions **flong** and **fshort** allow the programmer to influence the assembler whenever it is assembling an direct address the label of which contains a forward reference. By default the assembler will use the long direct address form. After an **fshort** pseudo instruction the assembler will generate the direct short address form. The occurrence of a **flong** pseudo instruction forces the direct long address form to be generated.

☛ The selected option, long or short direct addresses, is only valid until the next occurrence of a **flong**, **fshort** or segment control pseudo instruction.

4.9.5. NOOBJ - Suppress a.out Output.

noobj

Description:

The pseudo instruction **noobj** suppresses the generation of a a.out module.

4.10. Cross Reference Control

The pseudo instructions allow the programmer to select whether a cross reference listing shall be built up and printed at the end of the assembler listing. Default is **xrefon**.

xrefon

Description:
A cross reference listing is built up and printed.

xrefoff

Description:
A cross reference listing is suppressed.

5. MACRO OPERATIONS

A macro definition is a sequence of source statements that are saved and then assembled whenever needed through a macro call. A macro call consists of the occurrence of the macro name in the operation field of a statement. It usually includes parameters to be substituted for formal parameters in the macro code sequence so that code generated can vary with each assembly of the definition.

Use of a macro requires two steps, definition of the macro, and calling of the definition.

A definition consists of three parts: heading, body, and terminator.

heading — A macro definition is headed by a **macro** pseudo instruction stating the name of the macro. The heading optionally includes a **local** pseudo instruction identifying symbols local to the definition.

body — The body begins with the first statement in a definition that is not a **local** pseudo instruction or a comment line. The body consists of a series of symbolic instructions. All instructions other than **end** or another **macro** definition are legal within a definition. The assembler recognizes substitutable arguments in all fields of the source line. The macro argument ~0 however can only be used in the operation field for referring to the data size subparameter in an opcode or pseudo instruction. The arguments ~1 through ~9 can appear anywhere in a source line. Ten is the maximum number of arguments that can be handled by any macro definition. Macro calls may be nested up to ten levels deep.

terminator: An **endm** pseudo instruction terminates a macro definition.

5.1. ENDM - End Macro Definition.

endm

An **endm** pseudo instruction terminates the macro definition.

5.2. LOCAL - Local Symbols.

local <sym$_1$>, <sym$_2$>,..., <sym$_n$>

<sym$_1$>

List of local symbols. Symbols must be separated by commas. A blank terminates the list. The maximum number of local symbols is 10.

Description:

The **local** pseudo instruction, which lists symbols local to the definition optionally follows the **macro** pseudo instruction.

A symbol in the list is considered local to the macro; that is, it is known only within the macro definition. On each expansion of the macro, the assembler creates a new symbol for each local symbol and substitutes it for each occurrence of the local symbol in the definition. Thus invented symbols replace **local** named symbols wherever they appear in a macro definition in a manner similar to the way substitutable parameters are replaced.

5.3. MACRO - Macro Heading.

```
<m_name>    macro
```

<m_name>
A mandatory symbol that defines the name of the macro.

Description:
A **macro** pseudo instruction tells the assembler to place the instructions forming the body of the macro in a table of macro definitions for assembly upon call, and to place the macro name in the symbol table.

5.4. MACRO CALLS

A macro headed by a **macro** pseudo instruction can be called by an instruction in the following format:

```
<symbol>    <m_name>     <p1>, <p2>,..., <pn>
<symbol>    <m_name>.s   <p1>, <p2>,..., <pn>
<symbol>    <m_name>.b   <p1>, <p2>,..., <pn>
<symbol>    <m_name>.w   <p1>, <p2>,..., <pn>
<symbol>    <m_name>.l   <p1>, <p2>,..., <pn>
```

<symbol>
An optional location symbol.

<m_name>
Name of a previously defined macro. The (optional) size attribute substitutes macro parameter 0 (see above).

$<p_i>$

Parameter list composed of strings of characters. Parameters are separated by commas and terminated by a blank. Two consecutive commas constitute a null parameter.

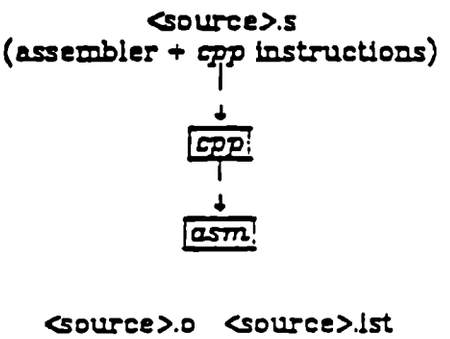
If null parameters are interspersed with non-null parameters, the correct positions must be established with commas. When the list terminates before the last possible parameter, all remaining parameters are considered null.

When the first character of a parameter is a left angle bracket (<), the assembler considers all the characters between it and the matching right angular bracket (>) as one parameter. The assembler removes the outer pair of angle brackets before substituting the enclosed character string in a line. Embedded brackets must be properly paired. A bracketed item can contain blanks and commas.

6. HOW TO USE THE ASSEMBLER

The assembler has been designed as a two pass assembler and is written in Pascal. To run the assembler two input files (INPUT - assembler source, INSERT - accessed only by the **insert** pseudo instruction), two output files (OUTPUT - assembler listing, AOUT - a.out object module) and two work files (SCRATCH, XREFFIL) must be provided.

Under UNIX the shell procedure *as* runs the preprocessor *cpp* and the assembler *asm*. One or more assembler source files may be listed as arguments. There is no provision for assignment of INSERT since insertion can be achieved more conveniently by use of *cpp*. Following UNIX convention, AOUT is written on a file named <name>.o, while the assembler sources should be named <name>.s. Listings will be displayed to <name>.lst.



Example 1:

as mysource.s
reads mysource.s
displays the listing to mysource.lst, the object module to mysource.o

Example 2:

as mysource*.s
reads mysource0.s mysource1.s mysource3.s
files the listings to mysource0.lst mysource1.lst mysource3.lst
writes the object modules to mysource0.o mysource1.o mysource3.o

Should you experience any problems or encounter errors, please contact the author.

7. APPENDIX

Symbols and related address modes and relocation information

symbol type	textsegment	datasegment
absolute	DA (r_abs)	DA (r_abs)
text	PCD16 (r_abs) DA (r_text)	DA (r_text)
data	DA (r_data)	PCD16 (r_abs) DA (r_text)
bss	DA (r_bss)	DA (r_bss)
external	DA (r_ext)	DA (r_ext)

Notation:

DA direct addressing mode

PCD16 program counter relative addressing mode (generated if requested)

Related relocation information

- r_abs (absolute symbol)
- r_text (text symbol)
- r_data (data symbol)
- r_bss (bss symbol)
- r_ext (external symbol)

List of Error Messages

0:constant too large
1:character not defined for m 68000 assembler
2:character missing or not valid for constant
3:string too long or not terminated properly
4:entry point or external symbol multiply defined
5:entry point not defined
7:symbol cannot be used as a label
8:this operation needs a label
9:this operation does not allow a label
10:symbol multiply defined
12:symbol not defined
13:initial ifeq or ifne is missing or misplaced
14:the label of an if should not be used here
15:if conditions more than maxinest levels deep
16:symbol cannot be used as an opcode
17:size specification illegal or not allowed
18:macro expansion error
19:more than maxmlocal local parameters
20:at most one local pseudo per macrodefinition
21:initial macro definition missing or misplaced
22:macro calls more than maxmnest levels deep
23:> expected
24:do not nest macro definitions
25:opcode/macro or pseudocode missing
26:no such cross-reference option
28:operation needs one or more operands
29:address or data register expected
30:address register expected
31:bad termination of an expression
32:an expression cannot start with this symbol
33:an operand cannot start with this symbol
34:the count must be absolute for this pseudo
35:the count must be positive for this pseudo
36:the expression must be greater lc for org pseudo
38:displacements are restricted to byte or word size
40:argument(s) missing in expression
41:displacement is restricted to byte size
42:type conflict between address and program counter
43:expression too complicated, use equ or set pseudo to break it down
44:expression too large for size specified
45:forward reference not allowed for this pseudo
46:both arguments must be absolute for logical operations
47:.s option does not allow branch to next word
48:register specification expected
49:) expected
50:) or , expected
51:separator expected
52:no size specification for this operation
53:byte size specification not allowed
54:both arguments must be absolute for shift operations
55:string expected

List of Error Messages (continued)

56:too many operands are specified for this operation
57:both arguments must be absolute for * or /
58:this address mode is illegal for the opcode
59:address mode combination illegal for opcode
60:do not write a comment on this line
61:synchronization error between pass one and two
62:too many errors this instruction line
63:fail generated error, consult listing
64:syntax error in register list for movem
65:org argument is of illegal type
68:no code generation in bss segment
69:list options are: dc, if, macro, xopc, xpse, xreg
70:one argument must be absolute for add operation
71:illegal operand types for sub operation
72:a.out buffers exceeded

Ergänzungen zu MUNIX - Fortran 77

Norbert Bladt
PCS Gmbh
Pfälzer-Wald-Strasse 36

8000 München 90

1. Einleitung

An dieser Stelle werden verschiedene systemspezifische Eigenschaften des MUNIX Fortran-77 Systems beschrieben, die in [1] nicht dargestellt werden.

2. Hinweise zu einigen Statements

2.1 OPEN-statement

- Nach einem *OPEN* steht der "file-pointer" immer auf end-of-file. Soll von einem sequentiellen File gelesen werden, so muss daher nach dem *OPEN* ein *REWIND*-statement ausgeführt werden.

- Der Filename muss, wenn er nicht als Literal angegeben wird, als *CHARACTER*N* deklariert sein.

- Bei direct-access-Files muss die Recordlänge in Byte angegeben werden.

- Bei *status='new'* darf, im Gegensatz zu früheren Versionen, das angegebene File nicht existieren, bei *status='old'* muss es dagegen vorhanden sein.

2.2 READ/WRITE-statements

- Entgegen der früheren Darstellung in [1] können die Fileunits 0 - 18 verwendet werden. Davon sind vorbelegt :

  0 - stderr (standard error output) = Terminal. Die Ausgabe kann durch "2>errors" in der Kommandozeile auf ein File "errors" umgeleitet werden.

  5 - stdin (standard input) = Terminal. Die Eingabe kann durch "<input" in der Kommandozeile von einem File "input" erfolgen.

  6 - stdout (standard output) = Terminal. Die Ausgabe kann durch ">output" in der Kommandozeile auf ein File "output" umgeleitet werden.

- Interne Files (entspricht *ENCODE/DECODE* auf anderen Systemen) müssen

  a) bei nur einem Record der Länge N als

  ```
  CHARACTER*N record
  ```

  b) bei m Records (m > 1) der Länge N als

  ```
  CHARACTER*N record(m)
  ```

  deklariert werden.

- Eine geklammerte E/A-Liste ohne implizite DO-Schleife ist ein Syntax-Fehler (ist in Standard Fortran-77 nicht erlaubt).

- Ein *CHARACTER*-Wert, der listengesteuert ausgegeben wurde, kann nicht wieder eingelesen werden (lt. Standard Fortran-77 werden *CHARACTER* als ein Zeichen ausgegeben, aber nur geklammert in ' oder " eingelesen !).

2.3 DATA-statement

- In DATA-Anweisungen können für CHARACTER nur darstellbare ASCII-Zeichen oder Ersatzzeichen verwendet werden.

  Ersatzzeichen \0 (binär Null),
  \f (formfeed),
  \n (newline),
  \\ (\)
  siehe auch in [1] Punkt 2.9.

  Eine Oktal- oder Hexadezimal-Darstellung wie bei INTEGER-Werten ist nicht möglich, ist aber für spätere Versionen vorgesehen.

2.4 PARAMETER-statement

- Im Gegensatz zu *DATA*-Anweisungen besteht bei der *PARAMETER*-Anweisung durchaus die Möglichkeit, *CHARACTER*-Konstanten nicht darstellbare ASCII-Zeichen zuzuordnen.

  Beispiel

  PARAMETER (ETX = char(3))

  Durch diese Anweisung wird eine CHARACTER-Konstante ETX definiert, die den binären Wert 3 erhält.

3. Allgemeine Hinweise

3.1 COMMON-Blöcke

- COMMON-Blöcke dürfen nur länger als 32 kByte sein, falls man beim Übersetzen die Option -Il verwendet.

- COMMON-Blöcke müssen bei jeder Deklaration die gleiche Länge besitzen !

3.2 Feld-Deklarationen

- Felder dürfen nur länger als 32 kByte sein, wenn man beim Übersetzen die Option -Il verwendet.

3.3 Sonstiges

- Konversionen zwischen den Datentypen

CHARACTER-Werte	--> INTEGER-Werte	Konversion automatisch
INTEGER-Werte	--> CHARACTER-Werte	**char**-Function (intrinsic)
INTEGER*2-Werte	--> INTEGER*4-Werte	**int4**-Function
INTEGER*4-Werte	--> INTEGER*2-Werte	**int2**-Function

- Die UNIX-spezifische Verwendung des Zeichens "\" als Fluchtsymbol bedingt auch in Fortran-77 eine abweichende Handhabung dieses Zeichens. Anstelle eines "\" sind stets zwei ("\\") anzugeben. Dazu siehe auch in [1] Punkt 2.9 und Punkt 2.3 oben.

- EQUIVALENCE auf CHARACTER-Feldern dürfen nur bei ungeraden Indices beginnen, sonst kommt es zu Laufzeitfehlern (odd address error).

- Die **Fortran-Steuerzeichen** 1, 0, + und (space) werden bei der Ausgabe nicht interpretiert. Sie können jedoch mit dem Filter "fpr" (siehe Man. 1) auf dem Drucker oder dem Terminal ausgegeben werden.

- Die (Unter-) Programme die einen Aufruf der Funktion **inquire** enthalten, müssen bei einer Umstellung von MUNIX-Version 1.4/06 auf MUNIX-Version 1.5 neu übersetzt werden.

- Zum Laden von bereits übersetzten f77-Moduln (suffix .o) sollte wie beim Compilieren der Aufruf
  "f77 prog.o sub.o "
  verwendet werden. Damit werden dann automatisch alle notwendigen Bibliotheken dazu gebunden und ein lauffähiges Programm erstellt.

- Ab MUNIX-Version 1.5 hat sich die Bedeutung der f77-Option **-I4** verändert.
  Ohne Verwendung weiterer Optionen bedeutete **-I4**
  früher 4-Byte Integer , 4-Byte Indices
  jetzt 4-Byte Integer, 2-Byte Indices.
  Dies bedeutet, dass man statt wie früher **-I4** nun **-I1** verwenden muss.

4. Abschliessende Bemerkungen

Ich hoffe, dass Ihnen mit diesen Hinweisen die Arbeit mit dem CADMUS/MUNIX-Fortran-77 Compiler und Laufzeitsystem erleichtert wird. Hinweise Ihrerseits werden vom Autor gerne entgegengenommen.

Literatur :

[1] - S.I. Feldman, P.J. Weinberger,
A Portable Fortran 77 Compiler,
Bell Lab's Murray Hill, New Jersey 07974

Restrictions/unsolved problems on MUNIX - Fortran 77

Norbert Bladt
PCS Gmbh
Pfälzer-Wald-Strasse 36

8000 München 90

Unsolved Problems

There are several unsolved problems in the fortran-77 runtime-system.

1. **statement-functions**

   Using the archaic feature of **statement-functions** in fortran-77 errors can occur. To avoid such errors, use *normal* functions instead.

2. **DATA**-statement

   The error-message **overlapping initialisation** without any linenumber can occur during initialisation of *CHARACTER*-strings.

3. The option **-fN**

   Using the **-fN** option the error-message **out of registers** means that you have to simplify your arithmetic expressions.

Restrictions

assign-statement

A *FORMAT*-statement must be given before an **assign**-statement that assigns the label of the format statement to an integer variable.

Changes in MUNIX - Fortran 77

Norbert Bladt
PCS Gmbh
Pfälzer-Wald-Strasse 36

D - 8000 München 90

Changes in f77 (MUNIX-Version 1.4/06 to 1.5)

There are several changes in the f77 runtime-system.
These are

1. the **inquire**-function.

   The internal call of the **inquire**-function has changed,
   i. e. you have to compile at least the (sub-)programs
   which call this function.

2. **the option -I4.**

   Without any other options it means now only 4-byte-integer
   (not as earlier also 4-byte-subscipts).
   To run with 4-byte-subscipts you now have to choose the
   -Il (ell) option, i.e. you have to change your invocation
   of the f77-compiler e.g. from
   f77 -I4 prog.f
   into
   f77 -Il prog.f

3. the **open**-statement.

   The meaning of the argument "status" has been changed
   according to the standard definition of Fortran-77.
   If **status='new'** is given, the file must not exist, if
   status='old' is given the file must exist. Otherwise
   an error is reported.

A Portable Fortran 77 Compiler

S. I. Feldman

P. J. Weinberger

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The Fortran language has just been revised. The new language, known as Fortran 77, became an official American National Standard on April 3, 1978. We report here on a compiler and runtime system for the new extended language. This is believed to be the first complete Fortran 77 system to be implemented. This compiler is designed to be portable, to be correct and complete, and to generate code compatible with calling sequences produced by C compilers. In particular, this Fortran is quite usable on UNIX† systems. In this paper, we describe the language compiled, interfaces between procedures, and file formats assumed by the I/O system. An appendix describes the Fortran 77 language.

1 August 1978

†UNIX is a Trademark of Bell Laboratories.

A Portable Fortran 77 Compiler

S. I. Feldman
P. J. Weinberger
Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

The Fortran language has just been revised. The new language, known as Fortran 77, became an official American National Standard [1] on April 3, 1978. for the language, known as Fortran 77, is about to be published. Fortran 77 supplants 1966 Standard Fortran [2]. We report here on a compiler and run-time system for the new extended language. The compiler and computation library were written by SIF, the I/O system by PJW. We believe ours to be the first complete Fortran 77 system to be implemented. This compiler is designed to be portable to a number of different machines, to be correct and complete, and to generate code compatible with calling sequences produced by compilers for the C language [3]. In particular, it is in use on UNIX† systems. Two families of C compilers are in use at Bell Laboratories, those based on D. M. Ritchie's PDP-11 compiler[4] and those based on S. C. Johnson's portable C compiler [5]. This Fortran compiler can drive the second passes of either family. In this paper, we describe the language compiled, interfaces between procedures, and file formats assumed by the I/O system. We will describe implementation details in companion papers.

1.1. Usage

At present, versions of the compiler run on and compile for the PDP-11, the VAX-11/780, and CADMUS 9000 UNIX systems. For the command to run the compiler see f77(1).

1.2. Documentation Conventions

In running text, we write Fortran keywords and other literal strings in boldface lower case. Examples will be presented in lightface lower case. Names representing a class of values will be printed in italics.

1.3. Implementation Strategy

The compiler and library are written entirely in C. The compiler generates C compiler intermediate code. Since there are C compilers running on a variety of machines, relatively small changes will make this Fortran compiler generate code for any of them. Furthermore, this approach guarantees that the resulting programs are compatible with C usage. The runtime computational library is complete. The mathematical functions are computed to at least 63 bit precision. The runtime I/O library makes use of D. M. Ritchie's Standard C I/O package [8] for transferring data. With the few exceptions described below, only documented calls are used, so it should be relatively easy to modify to run on other operating systems.

---

†UNIX is a Trademark of Bell Laboratories.

2. LANGUAGE EXTENSIONS

Fortran 77 includes almost all of Fortran 66 as a subset. We describe the differences briefly in the Appendix. The most important additions are a character string data type, file-oriented input/output statements, and random access I/O. Also, the language has been cleaned up considerably.

In addition to implementing the language specified in the new Standard, our compiler implements a few extensions described in this section. Most are useful additions to the language. The remainder are extensions to make it easier to communicate with C procedures or to permit compilation of old (1966 Standard) programs.

2.1. Double Complex Data Type

The new type **double complex** is defined. Each datum is represented by a pair of double precision real variables. A double complex version of every **complex** built-in function is provided. The specific function names begin with **z** instead of **c**.

2.2. Internal Files

The Fortran 77 standard introduces "internal files" (memory arrays) and restricts their use to formatted sequential I/O statements.

2.3. Implicit Undefined statement

Fortran 66 has a fixed rule that the type of a variable that does not appear in a type statement is **integer** if its first letter is **i, j, k, l, m** or **n**, and **real** otherwise. Fortran 77 has an **implicit** statement for overriding this rule. As an aid to good programming practice, we permit an additional type, **undefined**. The statement

```
implicit undefined(a-z)
```

turns off the automatic data typing mechanism, and the compiler will issue a diagnostic for each variable that is used but does not appear in a type statement. Specifying the **–u** compiler flag is equivalent to beginning each procedure with this statement.

2.4. Recursion

Procedures may call themselves, directly or through a chain of other procedures.

2.5. Automatic Storage

Two new keywords are recognized, **static** and **automatic**. These keywords may appear as "types" in type statements and in **implicit** statements. Local variables are static by default; there is exactly one copy of the datum, and its value is retained between calls. There is one copy of each variable declared **automatic** for each invocation of the procedure. Automatic variables may not appear in **equivalence, data,** or **save** statements.

2.6. Source Input Format

The Standard expects input to the compiler to be in 72 column format: except in comment lines, the first five characters are the statement number, the next is the continuation character, and the next sixty-six are the body of the line. (If there are fewer than seventy-two characters

on a line, the compiler pads it with blanks; characters after the seventy-second are ignored).

In order to make it easier to type Fortran programs, our compiler also accepts input in variable length lines. An ampersand ("&") in the first position of a line indicates a continuation line; the remaining characters form the body of the line. A tab character in one of the first six positions of a line signals the end of the statement number and continuation part of the line; the remaining characters form the body of the line. A tab elsewhere on the line is treated as another kind of blank by the compiler.

In the Standard, there are only 26 letters — Fortran is a one-case language. Consistent with ordinary UNIX system usage, our compiler expects lower case input. By default, the compiler converts all upper case characters to lower case except those inside character constants. However, if the **-U** compiler flag is specified, upper case letters are not transformed. In this mode, it is possible to specify external names with upper case letters in them, and to have distinct variables differing only in case. Regardless of the setting of the flag, keywords will only be recognized in lower case.

2.7. Include Statement

The statement

```
include 'stuff'
```

is replaced by the contents of the file **stuff**. **includes** may be nested to a reasonable depth, currently ten.

2.8. Binary Initialization Constants

A **logical**, **real**, or **integer** variable may be initialized in a **data** statement by a binary constant, denoted by a letter followed by a quoted string. If the letter is **b**, the string is binary, and only zeroes and ones are permitted. If the letter is **o**, the string is octal, with digits **0–7**. If the letter is **z** or **x**, the string is hexadecimal, with digits **0–9**, **a–f**. Thus, the statements

```
integer a(3)
data a / b'1010', o'12', z'a' /
```

initialize all three elements of **a** to ten.

2.9. Character Strings

For compatibility with C usage, the following backslash escapes are recognized:

\n	newline
\t	tab
\b	backspace
\f	form feed
\0	null
\'	apostrophe (does not terminate a string)
\"	quotation mark (does not terminate a string)
\\	\
x	*x*, where *x* is any other character

Fortran 77 only has one quoting character, the apostrophe. Our compiler and I/O system recognize both the apostrophe (') and the double-

quote ("). If a string begins with one variety of quote mark, the other may be embedded within it without using the repeated quote or backslash escapes.

Every unequivalenced scalar local character variable and every character string constant is aligned on an **integer** word boundary. Each character string constant appearing outside a **data** statement is followed by a null character to ease communication with C routines.

2.10. Hollerith

Fortran 77 does not have the old Hollerith (*n***h**) notation, though the new Standard recommends implementing the old Hollerith feature in order to improve compatibility with old programs. In our compiler, Hollerith data may be used in place of character string constants, and may also be used to initialize non-character variables in **data** statements.

2.11. Equivalence Statements

As a very special and peculiar case, Fortran 66 permits an element of a multiply-dimensioned array to be represented by a singly-subscripted reference in **equivalence** statements. Fortran 77 does not permit this usage, since subscript lower bounds may now be different from 1. Our compiler permits single subscripts in **equivalence** statements, under the interpretation that all missing subscripts are equal to 1. A warning message is printed for each such incomplete subscript.

2.12. One-Trip DO Loops

The Fortran 77 Standard requires that the range of a **do** loop not be performed if the initial value is already past the limit value, as in

```
do 10 i = 2, 1
```

The 1966 Standard stated that the effect of such a statement was undefined, but it was common practice that the range of a **do** loop would be performed at least once. In order to accommodate old programs, though they were in violation of the 1966 Standard, the **–onetrip** compiler flag causes non-standard loops to be generated.

2.13. Commas in Formatted Input

The I/O system attempts to be more lenient than the Standard when it seems worthwhile. When doing a formatted read of non-character variables, commas may be used as value separators in the input record, overriding the field lengths given in the format statement. Thus, the format

```
(i10, f20.10, i4)
```

will read the record

```
–345,.05e–3,12
```

correctly.

2.14. Short Integers

On machines that support halfword integers, the compiler accepts declarations of type **integer*2**. (Ordinary integers follow the Fortran rules about occupying the same space as a REAL variable; they are assumed to be of C type **long int**; halfword integers are of C type short

int.) An expression involving only objects of type **integer*2** is of that type. Generic functions return short or long integers depending on the actual types of their arguments. If a procedure is compiled using the **-I2** flag, all small integer constants will be of type **integer*2**. If the precision of an integer-valued intrinsic function is not determined by the generic function rules, one will be chosen that returns the prevailing length (**integer*2** when the **-I2** command flag is in effect). When the **-I2** option is in effect, all quantities of type **logical** will be short. Note that these short integer and logical quantities do not obey the standard rules for storage association.

2.15. Additional Intrinsic Functions

This compiler supports all of the intrinsic functions specified in the Fortran 77 Standard. In addition, there are functions for performing bitwise Boolean operations (**or, and, xor,** and **not)** and for accessing the UNIX command arguments (**getarg** and **iargc**).

3. VIOLATIONS OF THE STANDARD

We know only thre ways in which our Fortran system violates the new standard:

3.1. Double Precision Alignment

The Fortran standards (both 1966 and 1977) permit **common** or **equivalence** statements to force a double precision quantity onto an odd word boundary, as in the following example:

```
real a(4)
double precision b,c

equivalence (a(1),b), (a(4),c)
```

Some machines (e.g., Honeywell 6000, IBM 360) require that double precision quantities be on double word boundaries; other machines (e.g., IBM 370), run inefficiently if this alignment rule is not observed. It is possible to tell which equivalenced and common variables suffer from a forced odd alignment, but every double precision argument would have to be assumed on a bad boundary. To load such a quantity on some machines, it would be necessary to use separate operations to move the upper and lower halves into the halves of an aligned temporary, then to load that double precision temporary; the reverse would be needed to store a result. We have chosen to require that all double precision real and complex quantities fall on even word boundaries on machines with corresponding hardware requirements, and to issue a diagnostic if the source code demands a violation of the rule.

3.2. Dummy Procedure Arguments

If any argument of a procedure is of type character, all dummy procedure arguments of that procedure must be declared in an **external** statement. This requirement arises as a subtle corollary of the way we represent character string arguments and of the one-pass nature of the compiler. A warning is printed if a dummy procedure is not declared **external.** Code is correct if there are no **character** arguments.

3.3. T and TL Formats

The implementation of the **t** (absolute tab) and **tl** (leftward tab) format codes is defective. These codes allow rereading or rewriting part of the record which has already been processed. (Section 6.3.2 in the Appendix.) The implementation uses seeks, so if the unit is not one which allows seeks, such as a terminal, the program is in error. (People who can make a case for using **tl** should let us know.) A benefit of the implementation chosen is that there is no upper limit on the length of a record, nor is it necessary to predeclare any record lengths except where specifically required by Fortran or the operating system.

4. INTER-PROCEDURE INTERFACE

To be able to write C procedures that call or are called by Fortran procedures, it is necessary to know the conventions for procedure names, data representation, return values, and argument lists that the compiled code obeys.

4.1. Procedure Names

On UNIX systems, the name of a common block or a Fortran procedure has an underscore appended to it by the compiler to distinguish it from a C procedure or external variable with the same user-assigned name. Fortran library procedure names have embedded underscores to avoid clashes with user-assigned subroutine names.

4.2. Data Representations

The following is a table of corresponding Fortran and C declarations:

Fortran	C
integer*2 x	short int x;
integer x	long int x;
logical x	long int x;
real x	float x;
double precision x	double x;
complex x	struct { float r, i; } x;
double complex x	struct { double dr, di; } x;
character*6 x	char x[6];

(By the rules of Fortran, **integer**, **logical**, and **real** data occupy the same amount of memory).

4.3. Return Values

A function of type **integer**, **logical**, **real**, or **double precision** declared as a C function that returns the corresponding type. A **complex** or **double complex** function is equivalent to a C routine with an additional initial argument that points to the place where the return value is to be stored. Thus,

```
complex function f( . . )
```

is equivalent to

```
f_(temp, . . .)
struct { float r, i; } *temp;
```

A character-valued function is equivalent to a C routine with two extra initial

arguments: a data address and a length. Thus,

```
character*15 function g( . . . )
```

is equivalent to

```
g_(result, length, . . .)
char result[ ];
long int length;
```

and could be invoked in C by

```
char chars[15];
. . .
g_(chars, 15L, . .  );
```

Subroutines are invoked as if they were integer-valued functions whose value specifies which alternate return to use. Alternate return arguments (statement labels) are not passed to the function, but are used to do an indexed branch in the calling procedure. (If the subroutine has no entry points with alternate return arguments, the returned value is undefined.) The statement

```
call nret(*1, *2, *3)
```

is treated exactly as if it were the computed **goto**

```
goto (1, 2, 3), nret( )
```

4.4. Argument Lists

All Fortran arguments are passed by address. In addition, for every argument that is of type character or that is a dummy procedure, an argument giving the length of the value is passed. (The string lengths are **long int** quantities passed by value). The order of arguments is then:

```
Extra arguments for complex and character functions
Address for each datum or function
A long int for each character or procedure argument
```

Thus, the call in

```
external f
character*7 s
integer b(3)
. . .
call sam(f, b(2), s)
```

is equivalent to that in

```
int f();
char s[7];
long int b[3];

sam_(f, &b[1], s, 0L, 7L);
```

Note that the first element of a C array always has subscript zero, but Fortran arrays begin at 1 by default. Fortran arrays are stored in column-major order, C arrays are stored in row-major order.

5. FILE FORMATS

5.1. Structure of Fortran Files

Fortran requires four kinds of external files: sequential formatted and unformatted, and direct formatted and unformatted. On UNIX systems, these are all implemented as ordinary files which are assumed to have the proper internal structure.

Fortran I/O is based on "records". When a direct file is opened in a Fortran program, the record length of the records must be given, and this is used by the Fortran I/O system to make the file look as if it is made up of records of the given length. In the special case that the record length is given as 1, the files are not considered to be divided into records, but are treated as byte-addressable byte strings; that is, as ordinary UNIX file system files. (A read or write request on such a file keeps consuming bytes until satisfied, rather than being restricted to a single record.)

The peculiar requirements on sequential unformatted files make it unlikely that they will ever be read or written by any means except Fortran I/O statements. Each record is preceded and followed by an integer containing the record's length in bytes.

The Fortran I/O system breaks sequential formatted files into records while reading by using each newline as a record separator. The result of reading off the end of a record is undefined according to the Standard. The I/O system is permissive and treats the record as being extended by blanks. On output, the I/O system will write a newline at the end of each record. It is also possible for programs to write newlines for themselves. This is an error, but the only effect will be that the single record the user thought he wrote will be treated as more than one record when being read or backspaced over.

5.2. Portability Considerations

The Fortran I/O system uses only the facilities of the standard C I/O library, a widely available and fairly portable package, with the following two nonstandard features: The I/O system needs to know whether a file can be used for direct I/O, and whether or not it is possible to backspace. Both of these facilities are implemented using the **fseek** routine, so there is a routine **canseek** which determines if **fseek** will have the desired effect. Also, the inquire statement provides the user with the ability to find out if two files are the same, and to get the name of an already opened file in a form which would enable the program to reopen it. (The UNIX operating system implementation attempts to determine the full pathname.) Therefore there are two routines which depend on facilities of the operating system to provide these two services. In any case, the I/O system runs on the PDP-11, VAX-11/780, and Interdata 8/32 UNIX systems.

5.3. Pre-Connected Files and File Positions

Units 5, 6, and 0 are preconnected when the program starts. Unit 5 is connected to the standard input, unit 6 is connected to the standard output, and unit 0 is connected to the standard error unit. All are connected for sequential formatted I/O.

All the other units are also preconnected when execution begins. Unit *n* is connected to a file named **fort.***n*. These files need not exist, nor will they be created unless their units are used without first executing an open. The default connection is for sequential formatted I/O.

The Standard does not specify where a file which has been explicitly opened for sequential I/O is initially positioned. In fact, the I/O system attempts to position the file at the end, so a **write** will append to the file and a **read** will result in an end-of-file indication. To position a file to its beginning, use a **rewind** statement. The preconnected units 0, 5, and 6 are positioned as they come from the program's parent process.

REFERENCES

1. *Sigplan Notices* **11**, No.3 (1976), as amended in X3J3 internal documents through "/90.1".
2. *USA Standard FORTRAN, USAS X3.9-1966*, New York: United States of America Standards Institute, March 7, 1966. Clarified in *Comm. ACM* **12**, 289 (1969) and *Comm. ACM* **14**, 628 (1971).
3. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs: Prentice-Hall (1978).
4. D. M. Ritchie, private communication.
5. S. C. Johnson, "A Portable Compiler: Theory and Practice", Proc. 5th ACM Symp. on Principles of Programming Languages (January 1978).
6. S. I. Feldman, "An Informal Description of EFL", internal memorandum.
7. B. W. Kernighan, "RATFOR — A Preprocessor for a Rational Fortran", *Bell Laboratories Computing Science Technical Report #55*, (January 1977).
8. D. M. Ritchie, private communication.

APPENDIX. Differences Between Fortran 66 and Fortran 77

The following is a very brief description of the differences between the 1966 [2] and the 1977 [1] Standard languages. We assume that the reader is familiar with Fortran 66. We do not pretend to be complete, precise, or unbiased, but plan to describe what we feel are the most important aspects of the new language. At present the only current information on the 1977 Standard is in publications of the X3J3 Subcommittee of the American National Standards Institute. The following information is from the "/92" document. This draft Standard is written in English rather than a meta-language, but it is forbidding and legalistic. No tutorials or textbooks are available yet.

1. Features Deleted from Fortran 66

1.1. Hollerith

All notions of "Hollerith" (*n*h) as data have been officially removed, although our compiler, like almost all in the foreseeable future, will continue to support this archaism.

1.2. Extended Range

In Fortran 66, under a set of very restrictive and rarely-understood conditions, it is permissible to jump out of the range of a **do** loop, then jump back into it. Extended range has been removed in the Fortran 77 language. The restrictions are so special, and the implementation of extended range is so unreliable in many compilers, that this change really counts as no loss.

2. Program Form

2.1. Blank Lines

Completely blank lines are now legal comment lines.

2.2. Program and Block Data Statements

A main program may now begin with a statement that gives that program an external name:

```
program work
```

Block data procedures may also have names.

```
block data stuff
```

There is now a rule that only *one* unnamed block data procedure may appear in a program. (This rule is not enforced by our system.) The Standard does not specify the effect of the program and block data names, but they are clearly intended to aid conventional loaders.

2.3. ENTRY Statement

Multiple entry points are now legal. Subroutine and function subprograms may have additional entry points, declared by an **entry** statement with an optional argument list.

```
entry extra(a, b, c)
```

Execution begins at the first statement following the **entry** line. All variable declarations must precede all executable statements in the procedure. If the procedure begins with a **subroutine** statement, all entry

points are subroutine names. If it begins with a **function** statement, each entry is a function entry point, with type determined by the type declared for the entry name. If any entry is a character-valued function, then all entries must be. In a function, an entry name of the same type as that where control entered must be assigned a value. Arguments do not retain their values between calls. (The ancient trick of calling one entry point with a large number of arguments to cause the procedure to "remember" the locations of those arguments, then invoking an entry with just a few arguments for later calculation, is still illegal. Furthermore, the trick doesn't work in our implementation, since arguments are not kept in static storage.)

2.4. DO Loops

do variables and range parameters may now be of integer, real, or double precision types. (The use of floating point **do** variables is very dangerous because of the possibility of unexpected roundoff, and we strongly recommend against their use). The action of the **do** statement is now defined for all values of the **do** parameters. The statement

```
do 10 i = l, u, d
```

performs $\max(0, \lfloor (u-l)/d \rfloor)$ iterations. The **do** variable has a predictable value when exiting a loop: the value at the time a **goto** or **return** terminates the loop; otherwise the value that failed the limit test.

2.5. Alternate Returns

In a **subroutine** or subroutine **entry** statement, some of the arguments may be noted by an asterisk, as in

```
subroutine s(a, *, b, *)
```

The meaning of the "alternate returns" is described in section 5.2 of the Appendix.

3. Declarations

3.1. CHARACTER Data Type

One of the biggest improvements to the language is the addition of a character-string data type. Local and common character variables must have a length denoted by a constant expression:

```
character*17 a, b(3,4)
character*(6+3) c
```

If the length is omitted entirely, it is assumed equal to 1. A character string argument may have a constant length, or the length may be declared to be the same as that of the corresponding actual argument at run time by a statement like

```
character*(*) a
```

(There is an intrinsic function **len** that returns the actual length of a character string). Character arrays and common blocks containing character variables must be packed: in an array of character variables, the first character of one element must follow the last character of the preceding element, without holes.

3.2. IMPLICIT Statement

The traditional implied declaration rules still hold: a variable whose name begins with **i, j, k, l, m,** or **n** is of type **integer**, other variables are of type **real**, unless otherwise declared. This general rule may be overridden with an **implicit** statement:

```
implicit real(a-c,g), complex(w-z), character*(17) (s)
```

declares that variables whose name begins with an **a ,b, c,** or **g** are **real**, those beginning with **w, x, y,** or **z** are assumed **complex**, and so on. It is still poor practice to depend on implicit typing, but this statement is an industry standard.

3.3. PARAMETER Statement

It is now possible to give a constant a symbolic name, as in

```
parameter (x=17, y=x/3, pi=3.14159d0, s='hello')
```

The type of each parameter name is governed by the type of the constant expressions. The right side of each equal sign must be a constant expression (an expression made up of constants, operators, and already defined parameters).

3.4. Array Declarations

Arrays may now have as many as seven dimensions. (Only three were permitted in 1966). The lower bound of each dimension may be declared to be other than 1 by using a colon. Furthermore, an adjustable array bound may be an integer expression involving constants, arguments, and variables in **common**.

```
real a(-5:3, 7, m:n), b(n+1:2*n)
```

The upper bound on the last dimension of an array argument may be denoted by an asterisk to indicate that the upper bound is not specified:

```
integer a(5, *), b(*), c(0:1, -2:*)
```

3.5. SAVE Statement

A poorly known rule of Fortran 66 is that local variables in a procedure do not necessarily retain their values between invocations of that procedure. At any instant in the execution of a program, if a common block is declared neither in the currently executing procedure nor in any of the procedures in the chain of callers, all of the variables in that common block also become undefined. (The only exceptions are variables that have been defined in a **data** statement and never changed). These rules permit overlay and stack implementations for the affected variables. Fortran 77 permits one to specify that certain variables and common blocks are to retain their values between invocations. The declaration

```
save a, /b/, c
```

leaves the values of the variables **a** and **c** and all of the contents of common block **b** unaffected by a return. The simple declaration

```
save
```

has this effect on all variables and common blocks in the procedure. A

common block must be saved in every procedure in which it is declared if the desired effect is to occur.

3.6. INTRINSIC Statement

All of the functions specified in the Standard are in a single category, "intrinsic functions", rather than being divided into "intrinsic" and "basic external" functions. If an intrinsic function is to be passed to another procedure, it must be declared **intrinsic**. Declaring it **external** (as in Fortran 66) causes a function other than the built-in one to be passed.

4. Expressions

4.1. Character Constants

Character string constants are marked by strings surrounded by apostrophes. If an apostrophe is to be included in a constant, it is repeated:

```
'abc'
'ain''t'
```

There are no null (zero-length) character strings in Fortran 77. Our compiler has two different quotation marks, " ' "' and " " ". (See Section 2.9 in the main text.)

4.2. Concatenation

One new operator has been added, character string concatenation, marked by a double slash ("//"). The result of a concatenation is the string containing the characters of the left operand followed by the characters of the right operand. The strings

```
'ab' // 'cd'
'abcd'
```

are equal. The strings being concatenated must be of constant length in all concatenations that are not the right sides of assignments. (The only concatenation expressions in which a character string declared adjustable with a "*(*)" modifier or a substring denotation with nonconstant position values may appear are the right sides of assignments).

4.3. Character String Assignment

The left and right sides of a character assignment may not share storage. (The assumed implementation of character assignment is to copy characters from the right to the left side.) If the left side is longer than the right, it is padded with blanks. If the left side is shorter than the right, trailing characters are discarded.

4.4. Substrings

It is possible to extract a substring of a character variable or character array element, using the colon notation:

```
a(i,j) (m:n)
```

is the string of $(n-m+1)$ characters beginning at the m^{th} character of the character array element a_{ij}. Results are undefined unless $m \leq n$. Substrings may be used on the left sides of assignments and as procedure actual arguments.

4.5. Exponentiation

It is now permissible to raise real quantities to complex powers, or complex quantities to real or complex powers. (The principal part of the logarithm is used). Also, multiple exponentiation is now defined:

```
a**b**c = a ** (b**c)
```

4.6. Relaxation of Restrictions

Mixed mode expressions are now permitted. (For instance, it is permissible to combine integer and complex quantities in an expression.)

Constant expressions are permitted where a constant is allowed, except in **data** statements. (A constant expression is made up of explicit constants and **parameters** and the Fortran operators, except for exponentiation to a floating-point power). An adjustable dimension may now be an integer expression involving constants, arguments, and variables in B common..

Subscripts may now be general integer expressions; the old *cv±c* rules have been removed. **do** loop bounds may be general integer, real, or double precision expressions. Computed **goto** expressions and I/O unit numbers may be general integer expressions.

5. Executable Statements

5.1. IF-THEN-ELSE

At last, the if-then-else branching structure has been added to Fortran. It is called a "Block If". A Block If begins with a statement of the form

```
if (     ) then
```

and ends with an

```
end if
```

statement. Two other new statements may appear in a Block If. There may be several

```
else if(.  .) then
```

statements, followed by at most one

```
else
```

statement. If the logical expression in the Block If statement is true, the statements following it up to the next **elseif**, **else**, or **endif** are executed. Otherwise, the next **elseif** statement in the group is executed. If none of the sh: It: not found **elseif** conditions are true, control passes to the statements following the **else** statement, if any. (The **else** must follow all **elseifs** in a Block If. Of course, there may be Block Ifs embedded inside of other Block If structures). A case construct may be rendered

```
if (s .eq. 'ab') then
.
else if (s .eq. 'cd') then

else

end if
```

5.2. Alternate Returns

Some of the arguments of a subroutine call may be statement labels preceded by an asterisk, as in

```
call joe(j, *10, m, *2)
```

A return statement may have an integer expression, such as

```
return k
```

If the entry point has n alternate return (asterisk) arguments and if $1 \le k \le n$, the return is followed by a branch to the corresponding statement label; otherwise the usual return to the statement following the call is executed.

6. Input/Output

6.1. Format Variables

A format may be the value of a character expression (constant or otherwise), or be stored in a character array, as in

```
write(6, '(i5)') x
```

6.2. END=, ERR=, and IOSTAT= Clauses

A read or write statement may contain end=, err=, and iostat= clauses, as in

```
write(6, 101, err=20, iostat=a(4))
read(5, 101, err=20, end=30, iostat=x)
```

Here 5 and 6 are the *units* on which the I/O is done, 101 is the statement number of the associated format, 20 and 30 are statement numbers, and a and x are integers. If an error occurs during I/O, control returns to the program at statement 20. If the end of the file is reached, control returns to the program at statement 30. In any case, the variable referred to in the iostat= clause is given a value when the I/O statement finishes. (Yes, the value is assigned to the name on the right side of the equal sign.) This value is zero if all went well, negative for end of file, and some positive value for errors.

6.3. Formatted I/O

6.3.1. Character Constants

Character constants in formats are copied literally to the output. Character constants cannot be read into.

```
write(6,'(i2," isn""t ",i1)') 7, 4
```

produces

```
 7 isn't 4
```

Here the format is the character constant

```
(i2,' isn"t ',i1)
```

and the character constant

```
 isn't
```

is copied into the output.

6.3.2. Positional Editing Codes

t, **tl**, **tr**, and **x** codes control where the next character is in the record. **tr***n* or ***n*x** specifies that the next character is *n* to the right of the current position. **tl***n* specifies that the next character is *n* to the left of the current position, allowing parts of the record to be reconsidered. **t***n* says that the next character is to be character number *n* in the record. (See section 3.4 in the main text.)

6.3.3. Colon

A colon in the format terminates the I/O operation if there are no more data items in the I/O list, otherwise it has no effect. In the fragment

```
x='("hello", :, " there", i4)'
write(6, x) 12
write(6, x)
```

the first **write** statement prints **hello there 12**, while the second only prints **hello**.

6.3.4. Optional Plus Signs

According to the Standard, each implementation has the option of putting plus signs in front of non-negative numeric output. The **sp** format code may be used to make the optional plus signs actually appear for all subsequent items while the format is active. The **ss** format code guarantees that the I/O system will not insert the optional plus signs, and the **s** format code restores the default behavior of the I/O system. (Since we never put out optional plus signs, **ss** and **s** codes have the same effect in our implementation.)

6.3.5. Blanks on Input

Blanks in numeric input fields, other than leading blanks will be ignored following a **bn** code in a format statement, and will be treated as zeros following a **bz** code in a format statement. The default for a unit may be changed by using the **open** statement. (Blanks are ignored by default.)

6.3.6. Unrepresentable Values

The Standard requires that if a numeric item cannot be represented in the form required by a format code, the output field must be filled with asterisks. (We think this should have been an option.)

6.3.7. Iw.m

There is a new integer output code, **i***w.m*. It is the same as **i***w*, except that there will be at least *m* digits in the output field, including, if necessary, leading zeros. The case **i***w*.0 is special, in that if the value being printed is 0, the output field is entirely blank. **i***w*.1 is the same as **i***w*.

6.3.8. Floating Point

On input, exponents may start with the letter **E**, **D**, **e**, or **d**. All have the same meaning. On output we always use **e**. The **e** and **d** format codes also have identical meanings. A leading zero before the decimal point in **e** output without a scale factor is optional with the implementation. (We do not print it.) There is a **g***w.d* format code which is the same as **e***w.d*

and *fw.d* on input, but which chooses **f** or **e** formats for output depending. on the size of the number and of *d*.

6.3.9. "A" Format Code

A codes are used for character values. **a***w* use a field width of *w*, while a plain **a** uses the length of the character item.

6.4. Standard Units

There are default formatted input and output units. The statement

```
read 10, a, b
```

reads from the standard unit using format statement 10. The default unit may be explicitly specified by an asterisk, as in

```
read(*, 10) a,b
```

Similarly, the standard output units is specified by a **print** statement or an asterisk unit:

```
print 10
write(*, 10)
```

6.5. List-Directed Formatting

List-directed I/O is a kind of free form input for sequential I/O. It is invoked by using an asterisk as the format identifier, as in

```
read(6, *) a,b,c
```

On input, values are separated by strings of blanks and possibly a comma. Values, except for character strings, cannot contain blanks. End of record counts as a blank, except in character strings, where it is ignored. Complex constants are given as two real constants separated by a comma and enclosed in parentheses. A null input field, such as between two consecutive commas, means the corresponding variable in the I/O list is not changed. Values may be preceded by repetition counts, as in

```
4*(3.,2.)  2*, 4*'hello'
```

which stands for 4 complex constants, 2 null values, and 4 string constants.

For output, suitable formats are chosen for each item. The values of character strings are printed; they are not enclosed in quotes, so they cannot be read back using list-directed input.

6.6. Direct I/O

A file connected for direct access consists of a set of equal-sized records each of which is uniquely identified by a positive integer. The records may be written or read in any order, using direct access I/O statements.

Direct access **read** and **write** statements have an extra argument, **rec=**, which gives the record number to be read or written.

```
read(2, rec=13, err=20) (a(i), i=1, 203)
```

reads the thirteenth record into the array **a**.

The size of the records must be given by an **open** statement (see below). Direct access files may be connected for either formatted or unformatted I/O.

6.7. Internal Files

Internal files are character string objects, such as variables or substrings, or arrays of type character. In the former cases there is only a single record in the file, in the latter case each array element is a record. The Standard includes only sequential formatted I/O on internal files. (I/O is not a very precise term to use here, but internal files are dealt with using **read** and **write**). There is no list-directed I/O on internal files. Internal files are used by giving the name of the character object in place of the unit number, as in

```
character*80 x
read(5,"(a)") x
read(x,"(i3,i4)") n1,n2
```

which reads a card image into **x** and then reads two integers from the front of it. A sequential **read** or **write** always starts at the beginning of an internal file.

(We also support a compatible extension, direct I/O on internal files. This is like direct I/O on external files, except that the number of records in the file cannot be changed.)

6.8. OPEN, CLOSE, and INQUIRE Statements

These statements are used to connect and disconnect units and files, and to gather information about units and files.

6.8.1. OPEN

The **open** statement is used to connect a file with a unit, or to alter some properties of the connection. The following is a minimal example.

```
open(1, file='fort.junk')
```

open takes a variety of arguments with meanings described below.

unit= a small non-negative integer which is the unit to which the file is to be connected. We allow, at the time of this writing, 0 through 18. If this parameter is the first one in the **open** statement, the **unit=** can be omitted.

iostat= is the same as in **read** or **write**.

err= is the same as in **read** or **write**.

file= a character expression, which when stripped of trailing blanks, is the name of the file to be connected to the unit. The filename should not be given if the **status=scratch**.

status= one of **old**, **new**, **scratch**, or **unknown**. If this parameter is not given, **unknown** is assumed. If **scratch** is given, a temporary file will be created. Temporary files are destroyed at the end of execution. If **new** is given, the file must not exist. If **old** is given, it must exist. The meaning of **unknown** is processor dependent; our system opens the file if it exists and creates it otherwise.

access= **sequential** (default) or **direct**, depending on whether the file is to be opened for sequential or direct I/O.

form= **formatted** or **unformatted**. If this parameter is not given, **formatted** is assumed for a sequential file and **unformatted** for a direct-access file.

recl= a positive integer specifying the record length of the direct access file being opened. We measure all record lengths in bytes. On UNIX† systems a record length of 1 has the special meaning explained in section 5.1 of the text.

blank= **null** or **zero**. This parameter has meaning only for formatted I/O. The default value is **null**. **zero** means that blanks, other than leading blanks, in numeric input fields are to be treated as zeros.

Opening a new file on a unit which is already connected has the effect of first closing the old file. Note that the file pointer is positioned at the end of a sequential file; for reading the file must be rewound.

6.8.2. CLOSE

close severs the connection between a unit and a file. The unit number must be given. The optional parameters are **iostat=** and **err=** with their usual meanings, and **status=** either **keep** or **delete**. Scratch files cannot be kept, otherwise **keep** is the default. **delete** means the file will be removed. A simple example is

```
close(3, err=17)
```

6.8.3. INQUIRE

The **inquire** statement gives information about a unit ("inquire by unit") or a file ("inquire by file"). Simple examples are:

```
inquire(unit=3, namexx)
inquire(file='junk', number=n, exist=l)
```

file= a character variable specifies the file the **inquire** is about. Trailing blanks in the file name are ignored.

unit= an integer variable specifies the unit the **inquire** is about. Exactly one of **file=** or **unit=** must be used.

iostat=, **err=** are as before.

exist= a logical variable. The logical variable is set to **.true.** if the file or unit exists and is set to **.false.** otherwise.

opened= a logical variable. The logical variable is set to **.true.** if the file is connected to a unit or if the unit is connected to a file, and it is set to .false. otherwise.

number= an integer variable to which is assigned the number of the unit connected to the file, if any.

named= a logical variable to which is assigned **.true.** if the file has a name, or **.false.** otherwise.

---

†UNIX is a Trademark of Bell Laboratories.

name= a character variable to which is assigned the name of the file (inquire by file) or the name of the file connected to the unit (inquire by unit). The name will be the full name of the file.

access= a character variable to which will be assigned the value **'sequential'** if the connection is for sequential I/O, **'direct'** if the connection is for direct I/O. The value becomes undefined if there is no connection.

sequential= a character variable to which is assigned the value **'yes'** if the file could be connected for sequential I/O, **'no'** if the file could not be connected for sequential I/O, and **'unknown'** if we can't tell.

direct= a character variable to which is assigned the value **'yes'** if the file could be connected for direct I/O, **'no'** if the file could not be connected for direct I/O, and **'unknown'** if we can't tell.

form= a character variable to which is assigned the value **'formatted'** if the file is connected for formatted I/O, or **'unformatted'** if the file is connected for unformatted I/O.

formatted= a character variable to which is assigned the value **'yes'** if the file could be connected for formatted I/O, **'no'** if the file could not be connected for formatted I/O, and **'unknown'** if we can't tell.

unformatted= a character variable to which is assigned the value **'yes'** if the file could be connected for unformatted I/O, **'no'** if the file could not be connected for unformatted I/O, and **'unknown'** if we can't tell.

recl= an integer variable to which is assigned the record length of the records in the file if the file is connected for direct access.

nextrec= an integer variable to which is assigned one more than the number of the last record read from a file connected for direct access.

blank= a character variable to which is assigned the value **'null'** if null blank control is in effect for the file connected for formatted I/O, **'zero'** if blanks are being converted to zeros and the file is connected for formatted I/O.

The gentle reader will remember that the people who wrote the standard probably weren't thinking of his needs. Here is an example. The declarations are omitted.

```
open(1, file="/dev/console")
```

On a UNIX system this statement opens the console for formatted sequential I/O. An **inquire** statement for either unit 1 or file "/dev/console" would reveal that the file exists, is connected to unit 1, has a name, namely "/dev/console", is opened for sequential I/O, could be connected for sequential I/O, could not be connected for direct I/O (can't seek), is connected for formatted I/O, could be connected for formatted I/O, could not be connected for unformatted I/O (can't seek), has neither a record length nor a next record number, and is ignoring blanks in numeric fields.

In the UNIX system environment, the only way to discover what permissions you have for a file is to open it and try to read and write it. The **err=** parameter will return system error numbers. The **inquire** statement does not give a way of determining permissions.

A Tutorial Introduction

to ADB

The debugging program ADB provides capabilities to examine "core" and other program files in a varity of formats, run programs with embedded breakpoints and patch files.

ADB is an indispensable but complex tool for debugging crashed programs. This document provides an introduction to ADB with examples of its use. It explains the various formatting options, techniques for debugging C programs, examples of printing file system information and patching.

J. F. Maranzano
S. R. Bourne
May 1977

Bell Laboratories
Murray Hill, New Jersey 07974

Author's initials: MU

Trademarks:

MUNIX, CADMUS	for PCS
DEC, PDP	for DEC
UNIX	for Bell Laboratories

1. Introduction

ADB is a debugging program that is available on UNIX. It provides capabilities to look at "core" files resulting from aborted programs, print output in a variety of formats, patch files, and run programs with embedded breakpoints. This document provides examples of the more useful features of ADB. The reader is expected to be familiar with the basic commands on UNIX † ,with the C language, and with References 1, 2 and 3.

2. A Quick Survey

2.1. Invocation

ADB is invoked as:

```
adb objfile corefile
```

where `objfile` is an executable UNIX file and `corefile` is a core image file. Many times this will look like:

```
adb a.out core
```

or more simply:

```
adb
```

where the defaults are `a.out` and `core` respectively. The filename minus (−) means ignore this argument as in:

```
adb - core
```

ADB has requests for examining locations in either file. The `?` request examines the contents of `objfile`, the `/` request examines the `corefile`. The general form of these requests is:

```
address ? format
```

or

```
address / format
```

2.2. Current Address

ADB maintains a current address, called dot, similar in function to the current pointer in the UNIX editor. When an address is entered, the current address is set to that location, so that:

```
#126?i
```

sets dot to hex 126 and prints the instruction at that address. The request:

†UNIX is a Trademark of Bell Laboratories.

```
.,10/d
```

prints 10 decimal numbers starting at dot. Dot ends up referring to the address of the last item printed. When used with the ? or / requests, the current address can be advanced by typing newline; it can be decremented by typing ^.

Addresses are represented by expressions. Expressions are made up from decimal, octal, and hexadecimal integers, and symbols from the program under test. These may be combined with the operators +, −, *, % (integer division), & (bitwise and), | (bitwise inclusive or), # (round up to the next multiple), and ~ (not) (internal arithmetic uses 32 bits). When typing a symbolic address for a C program, the user can type name or _name; ADB will recognize both forms.

2.3. Formats

To print data, a user specifies a collection of letters and characters that describe the format of the printout. Formats are "remembered" in the sense that typing a request without one will cause the new printout to appear in the previous format. The following are the most commonly used format letters.

b	one byte in octal
c	one byte as a character
o	one word in octal
d	one word in decimal
f	two words in floating point
i	MC68000 instruction
s	a null terminated character string
a	the value of dot
u	one word as unsigned integer
n	print a newline
r	print a blank space
^	backup dot

(Format letters are also available for "long" values, for example, 'D' for long decimal, 'X' for long hex, and 'F' for double floating point.) For other formats see the ADB manual.

2.4. General Request Meanings

The general form of a request is:

```
address,count command modifier
```

which sets 'dot' to address and executes the command count times.

The following table illustrates some general ADB command meanings:

Command	Meaning
?	Print contents from a.out file
/	Print contents from core file
=	Print value of "dot"
:	Breakpoint control
$	Miscellaneous requests
;	Request separator
!	Escape to shell

ADB catches signals, so a user cannot use a quit signal to exit from ADB. The request $q or $Q (or crtl-Z) must be used to exit from ADB.

3. Debugging C Programs

3.1. Debugging A Core Image

Consider the C program in Figure 1. The program is used to illustrate a common error made by C programmers. The object of the program is to change the lower case "t" to upper case in the string pointed to by charp and then write the character string to the file indicated by argument 1. The bug shown is that the character "T" is stored in the pointer charp instead of the string pointed to by charp. Executing the program produces a core file because of an out of bounds memory reference.

ADB is invoked by:

```
adb a.out core
```

The first debugging request:

```
$c
```

is used to give a C backtrace through the subroutines called. As shown in Figure 2 only one function (main) was called and the arguments argc and argv have hex values #2 and #efff4e respectively. Both of these values look reasonable; #2 = two arguments, #efff4e = address on stack of parameter vector.
The next request:

```
$C
```

is used to give a C backtrace plus an interpretation of all the local variables in each function and their values in hex. Note that of the value for cc only the first byte is significant, because cc is declared as char. Were it declared as short, only the first two bytes would be significant. If however cc would have been declared as register char or register short, the name cc would have been prefixed with a '<', the shown value would be the value of the register, and the significant byte or word would be the last byte or word of the value!

The next request:

```
$r
```

prints out the registers including the program counter and an interpretation of the instruction at that location.

The request:

```
$e
```

prints out the values of all external variables.

A map exists for each file handled by ADB. The map for the a.out file is referenced by ? whereas the map for core file is referenced by /. Furthermore, a

good rule of thumb is to use ? for instructions and / for data when looking at programs. To print out information about the maps type:

```
$m
```

This produces a report of the contents of the maps. More about these maps later.

In our example, it is useful to try to see the contents of the string pointed to by charp. This is done by:

```
*charp/s
```

which says use charp as a pointer in the core file and print the information as a character string. This gives the message 'data address not found', because #55 is not a valid data address. Using ADB similarly, we could print information about the arguments to a function. The request:

```
main.argc/d
```

prints the decimal core image value of the argument argc in the function main. The request:

```
*main.argv/3X
```

prints the long hex values of the three consecutive pointers pointed to by argv in the function main. Note that these values are the addresses of the arguments to main. Therefore:

```
#efff7a/s
```

prints the ASCII value of the first argument. Another way to print this value would have been

```
*"/s
```

The " means ditto which remembers the last address typed, in this case main.argc the * instructs ADB to use the address field of the core file as a pointer.

The request:

```
.=X
```

prints the current address (not its contents) in long hex which has been set to the address of the first argument. The current address, dot, is used by ADB to "remember" its current location. It allows the user to reference locations relative to the current address, for example:

```
.-10/d
```

3.2. Multiple Functions

Consider the C program illustrated in Figure 3. This program calls functions f, g, and h until the stack is exhausted and a core image is produced. This example is a bit dangerous. On a system with a 68000, which does not generally recover from stack overflows, the program will sooner or later crash. With a 68010, however, the stack may grow very very large!

Again you can enter the debugger via:

```
adb
```

which assumes the names a.out and core for the executable file and core image file respectively. The request:

```
$c
```

will fill a page of backtrace references to f, g, and h. Figure 4 shows an abbreviated list (typing Ctrl-C will terminate the output and bring you back to ADB request level).

The request:

```
,5$C
```

prints the five most recent activations.

Notice that each function (f, g, h) has a counter of the number of times it was called.

The request:

```
fcnt/d
```

prints the decimal value of the counter for the function f. Similarly gcnt and hcnt could be printed. To print the value of an automatic variable, for example the decimal value of x in the last call of the function h, type:

```
h.x/d
```

It is not possible to print stack frames other than the most recent activation of a function. Therefore, a user can print everything with $C or the occurrence of a variable in the most recent call of a function. It is possible with the $C request, however, to print the stack frame starting at some address as address$C.

3.3. Setting Breakpoints

Consider the C program in Figure 5. This program, which changes tabs into blanks, is adapted from Software Tools by Kernighan and Plauger, pp. 18-27

We will run this program under the control of ADB (see Figure 6) by:

```
adb a.out -
```

Breakpoints are set in the program as:

```
address:b [request]
```

The requests:

```
settab+4:b
fopen+4:b
getc+4:b
tabpos+4:b
```

set breakpoints at the start of these functions, except for getc, which is called as a function, but in reality a macro defined in <stdio.h>. With the C compiler option -g, C generates statement labels. So it would be possible to plant a breakpoint at line 32 with the command

```
L32:b
```

The above addresses are entered as

```
symbol+4
```

so that they will appear in any C backtrace since the first instruction of each function is the 68000 LINK instruction. If you want to see the correct values of the register variables, if any, in the topmost procedure, you have to set the breakpoint after the MOVEM instruction which comes second. Note that some of the functions are from the C library.

To print the location of breakpoints one types:

```
$b
```

The display indicates a count field. A breakpoint is bypassed count −1times before causing a stop. The command field indicates the ADB requests to be executed each time the breakpoint is encountered. In our example no command fields are present.

By displaying the original instructions at the function settab we see that the breakpoint is set after the LINK instruction. We can display the instructions using the ADB request:

```
settab,5?ia
```

This request displays five instructions starting at settab with the addresses of each location displayed. Another variation is:

```
settab,5?i
```

which displays the instructions with only the starting address.

Notice that we accessed the addresses from the a.out file with the ? command. In general when asking for a printout of multiple items, ADB will advance the current address the number of bytes necessary to satisfy the request; in the above example five instructions were displayed and the current address was advanced 20 (decimal) bytes.

To run the program one simply types:

```
:r
```

To delete a breakpoint, for instance the entry to the function settab, one types:

```
settab+4:d
```

To continue execution of the program from the breakpoint type:

```
:c
```

Once the program has stopped (in this case at the breakpoint for fopen), ADB requests can be used to display the contents of memory. For example:

```
$C
```

to display a stack trace, or:

```
tabs,3/8d
```

to print three lines of 8 locations each from the array called tabs. By this time (at location fopen) in the C program, settab has been called and should have set a one in every eighth location of tabs.

3.4. Advanced Breakpoint Usage

We recompile the program, this time with the -g and -L option:

```
cc -g -L figure5.c
```

The -g option will add symbols Li to the symbol table of the program which point to the code for line i. The -L option adds code to the program itself, that writes the current line number into the stackframe. If you set a breakpoint to Li, than the code for writing i into the stackframe will probably be executed next.

Now we start adb again, setting breakpoints to line 24, after the getc and just in front of the switch statement, and to the start of tabpos. See figure 7. The first breakpoint shall be executed three times, each time displaying the value of c as a character. When the tabpos breakpoint is reached, we want to have a full dump of the topmost stackframe. The file "data" contains the text "This<tab>is<tab>a...".

```
adb a.out -
L24,3:b main.c-1/C
tabpos+4:b ,1$C
:r
```

The program starts, displaying the value of c three times, and then stops. When we continue the program with:

```
:c
```

the L24 breakpoint is executed two more times, then we hit our breakpoint at tabpos since there is a tab following the "This" word of the data.

We see that tabpos is called with a value of 5. The displayed linenumber is incorrect because the code for storing the linenumber is not yet executed. Next we want to set a breakpoint at the end of the procedure. We could use the linenumber again, but for demonstration we are searching for the UNLK instruction at the end of the procedure, which has the opcode #4e5e, and set a breakpoint to it. The command

```
?I #4e5e
```

searches from the current value of dot, which is set (in this case) to tabpos+4, until it finds the value #4e5e. This value is reported to be at location tabpos+#2e, and dot is set to this address. With :b we set a breakpoint to dot, and with :c we continue. When we reach the breakpoint, we can look at the value tabpos returns, by displaying the contents of register D0. A final view of the active frames with $C shows us, that main is executing line 26 and tabpos line 49.

The UNIX quit and interrupt signals act on ADB itself rather than on the program being debugged. If such a signal occurs then the program being debugged is stopped and control is returned to ADB. The signal is saved by ADB and is passed on to the test program if:

```
:c
```

is typed. This can be useful when testing interrupt handling routines. The signal is not passed on to the test program if:

```
:c 0
```

is typed.

A breakpoint can be overwritten without first deleting the old breakpoint. For example:

```
settab+4:b  settab,5?ia; ptab/x
```

could be entered after typing the above requests.

3.5. Other Breakpoint Facilities

Arguments and change of standard input and output are passed to a program as:

```
:r  arg1  arg2 ... <infile  >outfile
```

This request kills any existing program under test and starts the a.out afresh.

The program being debugged can be single stepped by:

```
:s
```

If necessary, this request will start up the program being debugged and stop after executing the first instruction.

ADB allows a program to be entered at a specific address by typing:

```
address:r
```

The count field can be used to skip the first n breakpoints as:

```
,n:r
```

The request:

```
,n:c
```

may also be used for skipping the first n breakpoints when continuing a program.

A program can be continued at an address different from the breakpoint by:

```
address:c
```

The program being debugged runs as a separate process and can be killed by:

```
:k
```

4. Maps

UNIX knows several object file formats. These are used to tell the loader how to load the program file. File type 407 is the format of object modules generated by a C compiler invocation such as **cc -c pgm.c**. A 411 file is produced by the loader, when all references are resolved. This is the format of all executable files. ADB provides access to the program or module through a set of maps. To print the maps type:

```
$m
```

The b, e, and f fields are used by ADB to map addresses into file addresses. The "b" and "e" fields are the starting and ending locations for a segment, i.e. logical addresses. When an address A followed by ? or / is given, adb goes to the specified map, and sees if A lies in the interval [b1,e1] or [b2,e2]. If yes, it subtracts the corresponding b value from A and adds the corresponding f value, to obtain the physical address in the file. More formally, given an address, A, the location in the file (either **a.out** or **core**) is calculated as:

```
b1≤A≤e1 ⇒ file address = (A-b1)+f1
b2≤A≤e2 ⇒ file address = (A-b2)+f2
```

In the normal case, the first segment of the ? map corresponds to the a.out text, the second to the a.out data. The first segment of the / map corresponds to the core data segment, and the second segment to the core stack segment. A core file does not include the text of the crashed program. Note that data can be accessed in two ways. However, the a.out data segment is the data segment as it looked at load time. It also contains only the initialized data. The core data segment is the data at the time of the core dump. Note that the b2 value of the / map is the lower limit of the stack.

A user can access locations by using the ADB defined variables. The **$v** request prints the variables initialized by ADB (Figure 8):

b	base address of data segment
d	length of the data segment
e	entry point of the program
m	execution type (407,410,411)
s	length of the stack
t	length of the text

Use can be made of these variables by expressions such as:

```
<s
```

in the address field. Similarly the value of the variable can be changed by an assignment request such as:

```
02000>b
```

that sets **b** to octal 2000. These variables are useful to know if the file under examination is an executable or **core** image file.

ADB reads the header of the **core** image file to find the values for these variables. If the second file specified does not seem to be a **core** file, then standard values for identical mapping are used instead.

5. Advanced Usage

It is possible with ADB to combine formatting requests to provide elaborate displays. Below are several examples.

5.1. Formatted dump

The line:

```
<b,-1/4x4^8Cn
```

prints 4 hex words followed by their ASCII interpretation from the data space of the core image file. Broken down, the various request pieces mean:

<b The base address of the data segment.

<b,−1
Print from the base address to the end of file. A negative count is used here and elsewhere to loop indefinitely or until some error condition (like end of file) is detected.

The format **4x4^8Cn** is broken down as follows:

4x Print 4 hex locations.

4^ Backup the current address 4 locations (to the original start of the field).

8C Print 8 consecutive characters using an escape convention; each character in the range 0 to 037 is printed as @ followed by the corresponding character in the range 0140 to 0177. An @ is printed as @@.

n Print a newline.

The request:

```
<b,<d/4x4^8Cn
```

could have been used instead to allow the printing to stop at the end of the data segment (<d provides the data segment size in bytes).

The formatting requests can be combined with ADB's ability to read in a script to produce a core image dump script. ADB is invoked as:

```
adb a.out core < dump
```

to read in a script file, **dump**, of requests. An example of such a script is:

```
120$w
4095$s
$v
=3n
$m
=3n'C Stack Backtrace"
$C
=3n'C External Variables"
$e
=3n'Registers"
$r
0$s
=3n'Data Segment"
<b,-1/8xna
```

ADB attempts to print addresses as:

```
symbol + offset
```

The request **4095$s** sets the maximum permissible offset to the nearest symbolic address to 4095 (default 32767). The request = can be used to print literal strings. Thus, headings are provided in this **dump** program with requests of the form:

```
=3n'C Stack Backtrace"
```

that spaces three lines and prints the literal string. The request **$v** prints all non-zero ADB variables (see Figure 8). The request **0$s** sets the maximum offset

for symbol matches to zero thus suppressing the printing of symbolic labels in favor of hex values. Note that this is only done for the printing of the data segment. Try dumping the data segment without changing the maximum offset for a comparison. The request:

```
<b,-1/8xna
```

prints a dump from the base of the data segment to the end of file with a hex address field and eight hex numbers per line.

Figure 10 shows the results of some formatting requests on the C program of Figure 9.

5.2. Directory Dump

As another illustration (Figure 11) consider a set of requests to dump the contents of a directory (which is made up of an integer **inumber** followed by a 14 character name):

```
adb dir -
=n8t"Inum"8t"Name"
0,-1? u8t14cn
```

In this example, the **u** prints the **inumber** as an unsigned decimal integer, the **8t** means that ADB will space to the next multiple of 8 on the output line, and the **14c** prints the 14 character file name.

5.3. Ilist Dump

Similarly the contents of the **ilist** of a file system, (e.g. /dev/hk0, see **inode**(5)) could be dumped with the following set of requests:

```
adb /dev/hk0 -
1024,-1?"flags"8ton"links,uid,gid"8t3on",size"8tDn"addr"8t10Xn"times"8t3Y2na
```

In this example the dump begins at location 1024, since that is the start of an **ilist** within a 512 byte per block file system. The last access time, last modify time and creation time are printed with the **3Y** operator. Figure 11 shows portions of these requests as applied to a directory and file system.

5.4. Converting values

ADB may be used to convert values from one representation to another. For example:

```
072 = odx
```

will print

```
072     58      #3a
```

which are the octal, decimal and hexadecimal representations of 072 (octal). The format is remembered so that typing subsequent numbers will print them in the given formats. Character values may be converted similarly, for example:

```
'a' = co
```

prints

```
a       0141
```

It may also be used to evaluate expressions but be warned that all binary operators have the same precedence which is lower than that for unary operators.

6. Patching

Patching files with ADB is accomplished with the `write`, `w` or `W`, request (which is not like the `ed` editor write command). This is often used in conjunction with the `locate`, `l` or `L` request. In general, the request syntax for `l` and `w` are similar as follows:

```
?l value
```

The request `l` is used to match on two bytes, `L` is used for four bytes. The request `w` is used to write two bytes, whereas `W` writes four bytes. The `value` field in either `locate` or `write` requests is an expression. Therefore, decimal, octal and hex numbers, or character strings are supported.

In order to modify a file, ADB must be called as:

```
adb -w file1
```

When called with this option, `file1` is created if necessary and opened for both reading and writing.

For example, consider the C program shown in Figure 9. We can change the word "This" to "The " in the executable file for this program, `ex7`, by using the following requests:

```
adb -w ex7 -
<b?l 'Th'
?W 'The '
```

The request ?l starts at dot and stops at the first match of "Th" having set dot to the address of the location found. Note the use of ? to write to the `a.out` file.

More frequently the request will be typed as:

```
?l 'Th'; ?s
```

and locates the first occurrence of "Th" and print the entire string. Execution of this ADB request will set dot to the address of the "Th" characters.

As another example of the utility of the patching facility, consider a C program that has an internal logic flag. The flag could be set by the user through ADB and the program run. For example:

```
adb a.out -
:s arg1 arg2
flag/w 1
:c
```

The :s request is normally used to single step through a process or start a process in single step mode. In this case it starts **a.out** as a subprocess with arguments **arg1** and **arg2**. If there is a subprocess running ADB writes to it rather than to the file so the **w** request causes **flag** to be changed in the memory of the subprocess.

7. References

1. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," CACM, July, 1974.
2. B. W. Kernighan and D. M. Ritchie, The C Programming Language, Prentice-Hall, 1978.
3. K. Thompson and D. M. Ritchie, UNIX Programmer's Manual - 7th Edition, 1978.
4. B. W. Kernighan and P. J. Plauger, Software Tools, Addison-Wesley, 1976.

Figure 1: C program with pointer bug

```
#include <stdio.h>

char *charp = "this is a sentence.";
FILE *obuf;

main(argc,argv)
int argc;
char **argv;
{
        char    cc;

        if(argc < 2) {
                printf("Output file argument missing\n");
                exit(1);
        }

        if((obuf = fopen(argv[1],"w")) == NULL){
                printf("%s : not found\n", argv[1]);
                exit(1);
        }
        cc = 'X';       /* for DEMO purpose only */
        charp = 'T';    /* BUG: should be *charp = 'T';  !! */
        while(cc = *charp++)
                putc(cc,obuf);
        fclose(obuf);
}
```

Figure 2: ADB output for C program of Figure 1

```
adb a.out core
$c
~main(#0002,#00ef,#ff4e,#00ef,#ff5a) line 8


$C
~main(#0002,#00ef,#ff4e,#00ef,#ff5a) line 8
                _argc:          #000200ef
                _argv:          #00efff4e
                _cc:            #580000ef


$r
d0      #00800070                __iob+#0024
d1      #0080008c                _charp+#0008
d2      #00600e3e
d3      #00000000
d4      #00000000
d5      #00000000
d6      #00000000
d7      #00000000
a0      #00600e40
a1      #0080003a                _charp+#0036
a2      #00000054
a3      #00efff56
a4      #00000000
a5      #00000000
a6      #00efff18
a7      #00effefc
ns      #00021d51
ac      #00000054
ip      #1d520000
pc      #00600e4                 ~main+#0002
~main+#0002:    move.b  (a2),-28(a6)


$e
_environ:           #00efff5a
_charp:             #00000055
_pbuf:              #00800070
__iob:              #008005ca
__lastbuf:          #0080013c
__sibuf:            #00000000
__sobuf:            #00000000
__pfile:            #00000000
_errno:             #00000000
__ctype_:           #00202020
__currbrk:          #00800dd0
_end:               #00000000


$m
? map       'a.out'
b1 = #00600000      e1 = #0060126a      f1 = #0000001c
b2 = #00800000      e2 = #00800dd0      f2 = #00001286
/ map       'core'
b1 = #00800000      e1 = #00800e00      f1 = #00000c54
b2 = #00eff600      e2 = #00f00000      f2 = #00001a54


charp/X
_charp:         #00000055


*charp/s
#00000055:
```

```
data address not found

main.argc/d
#00efff20:      2

*main.argv/3X
#00efff4e:      #00efff7a       #00efff80       #00000000

#efff7a/s
#00efff7a:      a.out

*main.argv/3X
#00efff4e:      #00efff7a       #00efff80       #00000000

*"/s
#00efff7a:      a.out

.=X
                #00efff7a

$q
```

Figure 3: Multiple function C program for stack trace illustration

```
int     fcnt,gcnt,hcnt;
h(x,y)
{
        int hi; register int hr;
        hi = x+1;
        hr = x-y+1;
        hcnt++ ;
        hj:
        f(hr,hi);
}

g(p,q)
{
        int gi; register int gr;
        gi = q-p;
        gr = q-p+1;
        gcnt++ ;
        gj:
        h(gr,gi);
}

f(a,b)
{
        int fi; register int fr;
        fi = a+2*b;
        fr = a+b;
        fcnt++ ;
        fj:
        g(fr,fi);
}

main()
{
        f(1,1);
}
```

Figure 4: ADB output for C program of Figure 3

```
adb
$c
~h(#0040,#003f) line 0
~g(#0041,#0080) line 0
~f(#0002,#003f) line 0
~h(#003e,#003d) line 0
~g(#003f,#007c) line 0
~f(#0002,#003d) line 0
~h(#003c,#003b) line 0
~g(#003d,#0078) line 0
~f(#0002,#003b) line 0
~h(#003a,#0039) line 0
~g(#003b,#0074) line 0

HIT INTR KEY
adb

,5$C
~h(#0040,#003f) line 0
                _x:             #0040003f
                _y:             #003f003f
                _hi:            #00410000
               <_hr:            #00000002
~g(#0041,#0080) line 0
                _p:             #00410080
                _q:             #00800080
                _gi:            #003f0000
               <_gr:            #00000040
~f(#0002,#003f) line 0
                _a:             #0002003f
                _b:             #003f003f
                _fi:            #00800000
               <_fr:            #00000041
~h(#003e,#003d) line 0
                _x:             #003e003d
                _y:             #003d003d
                _hi:            #003f0000
               <_hr:            #00000002
~g(#003f,#007c) line 0
                _p:             #003f007c
                _q:             #007c007c
                _gi:            #003d0000
               <_gr:            #0000003e


fcnt/d
_fcnt:          32

gcnt/d
_gcnt:          32

hcnt/d
_hcnt:          31

h.x/d
#00eff826:      64

$q
```

Figure 5: C program to decode tabs

```
 1  #include <stdio.h>
 2  #define MAXLINE 80
 3  #define YES             1
 4  #define NO              0
 5  #define TABSP           8
 6
 7  char    input[] = "data";
 8  FILE    *ibuf;
 9  int     tabs[MAXLINE];
10
11  main()
12  {
13          int col, *ptab;
14          char c;
15
16          ptab = tabs;
17          settab(ptab);   /*Set initial tab stops */
18          col = 1;
19          if ((ibuf = fopen(input,"r")) == NULL) {
20                  printf("%s : not found\n",input);
21                  exit(1);
22          }
23          while((c = getc(ibuf)) != EOF) {
24                  switch(c) {
25                          case '\t':      /* TAB */
26                                  while(tabpos(col) != YES) {
27                                  putchar(' ');
28                                          col++ ;
29                                  }
30                                  break;
31                          case '\n':      /*NEWLINE */
32                                  putchar('\n');
33                                  col = 1;
34                                  break;
35                          default:
36                                  putchar(c);
37                                  col++ ;
38                  }
39          }
40  }
41
42  /* Tabpos return YES if col is a tab stop */
43  tabpos(col)
44  int col;
45  {
46          if(col > MAXLINE)
47                  return(YES);
48          else
49                  return(tabs[col]);
50  }
51
52  /* Settab - Set initial tab stops */
53  settab(tabp)
54  int *tabp;
55  {
56          int i;
57
58          for(i = 0; i<= MAXLINE; i++)
59                  (i%TABSP) ? (tabs[i] = NO) : (tabs[i] = YES);
60  }
```

Figure 6: ADB output for C program of Figure 5

```
adb a.out -
settab+4:b
fopen+4:b
getc+4:b
symbol not found

tabpos+4:b $b
breakpoints
count   bkpt            command
1       ~tabpos+#0004
1       _fopen+#0004
1       ~settab+#0004

settab,5?ia
~settab:        link    a6,#-28
~settab+#0004:  clr.w   -28(a6)
~settab+#0008:  cmpi.w  #80,-28(a6)
~settab+#000e:  bgt     ~settab+#0054
~settab+#0010:  move.w  -28(a6),d1
~settab+#0014:

settab,5?i
~settab:        link    a6,#-28
                clr.w   -28(a6)
                cmpi.w  #80,-28(a6)
                bgt     ~settab+#0054
                move.w  -28(a6),d1

:r
a.out: running
breakpoint      ~settab+#0004:  clr.w   -28(a6)

settab+4:d
:c
a.out: running
breakpoint      _fopen+#0004:   jsr     __findiop

$C
_fopen(#0000,#0004,#0000,#000a) line 0
~main(#0001,#00ef,#ff58,#00ef,#ff60) line 0
                _col:           #000100ef
                _ptab:          #000005ae
                _c:             #00000080

tabs,3/8d
_tabs:          1       0       0       0       0       0       0       0
                1       0       0       0       0       0       0       0
                1       0       0       0       0       0       0       0
```

Figure 7: ADB output for C program of Figure 5

```
adb a.out -
L24,3:b main.c/C
tabpos+4:b ,1$C
:r
a.out: running
#00effefe:      T
#00effefe:      h
#00effefe:      i
breakpoint      ~main+#00c4:    ext.w   d0


:c
a.out: running
#00effefe:      s
#00effefe:      @i
~tabpos(#0005) line -224
                _col:           #00050900
breakpoint      ~tabpos+#0004:  move.w  @47,-2(a6)


?l #4e5e
~tabpos+#002e

:b
:c
a.out: running
breakpoint      ~tabpos+#002e:  unlk    a6


<d0=x
                #0000


$C
~tabpos(#0005) line 49
                _col:           #00050900
~main(#0001,#00ef,#ff56,#00ef,#ff5e) line 26
                _col:           #000500ef
                _ptab:          #008005ae
                _c:             #09000080
```

Figure 8: ADB output for maps

```
adb cat core
$m
? map       'cat'
b1 = #00600000       e1 = #006017ee       f1 = #0000001c
b2 = #00800000       e2 = #008008d0       f2 = #0000180a
/ map       'core'
b1 = #00800000       e1 = #00800a00       f1 = #00000c54
b2 = #00eff600       e2 = #00f00000       f2 = #00001654


$v
variables
b = #00800000
d = #00000a00
e = #00600000
m = #00000109
s = #00000a00
t = #00001800
```

Figure 9: Simple C program for illustrating formatting and patching

```
char    str1[] = "This is a character string";
int     one    = 1;
int     number = 456;
long    lnum   = 1234;
float   fpt    = 1.25;
char    str2[] = "This is the second character string";
main()
{
        one = 2;
}
```

Figure 10: ADB output illustrating fancy formats

```
adb ex10 -
:s
stopped at      ~_entry:        link    a6,@-26

<b,6/8xna
_execargs:      #00ef   #fffc   #5468   #6973   #2069   #7320   #6120   #6368

_str1+#000c:    #6172   #6163   #7465   #7220   #7374   #7269   #6e67   #0000

_one:
_one:           #0001   #01c8   #0000   #04d2   #a000   #0041   #5468   #6973

_str2+#0004:    #2069   #7320   #7468   #6520   #7365   #636f   #6e64   #2063

_str2+#0014:    #6861   #7261   #6374   #6572   #2073   #7472   #696e   #6700

__iob:
__iob:          #0080   #0148   #0000   #0080   #0148   #0100   #0000   #0000

<b,10/4x4^8Cn
_execargs:      #00ef   #fffc   #5468   #6973   @'o@|This
                #2069   #7320   #6120   #6368    is a ch
                #6172   #6163   #7465   #7220   aracter
                #7374   #7269   #6e67   #0000   string@'@'

_one:           #0001   #01c8   #0000   #04d2   @'@s@aH@'@'@dR

_fpt:           #a000   #0041   #5468   #6973    @'@'AThis
                #2069   #7320   #7468   #6520    is the
                #7365   #636f   #6e64   #2063   second c
                #6861   #7261   #6374   #6572   haracter
                #2073   #7472   #696e   #6700    string@'

<b,10/4x4^8t8cna
_execargs:      #00ef   #fffc   #5468   #6973           o
|This
_str1+#0004:    #2069   #7320   #6120   #6368            is a ch
_str1+#000c:    #6172   #6163   #7465   #7220           aracter
_str1+#0014:    #7374   #7269   #6e67   #0000           string
_one:
_one:           #0001   #01c8   #0000   #04d2           .................HR
_fpt:
_fpt:           #a000   #0041   #5468   #6973            AThis
_str2+#0004:    #2069   #7320   #7468   #6520            is the
_str2+#000c:    #7365   #636f   #6e64   #2063           econd c
_str2+#0014:    #6861   #7261   #6374   #6572           haracter
_str2+#001c:    #2073   #7472   #696e   #6700            string

<b,10/2b8t^2cn
_execargs:      #0000   #00ef           o
                #00ff   #00fc
|

_str1:          #0054   #0068           Th
                #0069   #0073           is
                #0020   #0069            i
                #0073   #0020           s
                #0061   #0020           a
                #0063   #0068           ch
                #0061   #0072           ar
                #0061   #0063           ac

$Q
```

Figure 11: Directory and inode dumps

```
adb dir -
=nt"Inode"t"Name"
0,6?ut14cn

                Inode Name
#00000000:      167
                2       ..
                1049    renice
                334     shutdown.sh
                1391    cron
                163     ddate


adb /dev/hk0 -
1024,3?"flags"8ton"links,uid,gid"8t3on"size"8tDn"addr"8t10Xn"times"8t3Y2na
#00000400:      flags   0100000
                links,uid,gid   0       0       0
                size    0
                addr    #00000000       #00000000       #00000000       #00000000
                #00000000       #00000000       #00000000       #00000000
                #00000000       #00000000
                times   1984 Feb  8 17:42:14    1984 Feb  8 17:42:14    1984 Feb  8 17:42:14

#00000440:      flags   040777
                links,uid,gid   05      0       0
                size    2832
                addr    #00013300       #06b3000f       #42001399       #0018d500
                #1ffb0000       #00000000       #00000000       #00000000
                #00000000       #00000000
                times   1984 Mar  7 12:51:18    1984 Mar  7 15:29:53    1984 Mar  7 15:29:53

#00000480:      flags   0100755
                links,uid,gid   01      02      02
                size    16968
                addr    #00013a00       #01410001       #4800014f       #00015600
                #015d0001       #6400016b       #00017200       #01790001
                #80000000       #00000000
                times   1984 Mar  5 09:11:07    1983 Oct 12 13:00:28    1984 Feb  8 17:44:03
```

ADB Summary

Command Summary

a) formatted printing

?format print from a.out file according to format

/format print from core file according to format

=format print the value of dot

?w expr write expression into a.out file

/w expr write expression into core file

?l expr locate expression in a.out file

b) breakpoint and program control

:b set breakpoint at dot
:c continue running program
:d delete breakpoint
:k kill the program being debugged
:r run a.out file under ADB control
:s single step

c) miscellaneous printing

$b print current breakpoints
$c C stack trace
$e external variables
$f floating registers (not yet)
$m print ADB segment maps
$q exit from ADB
$r general registers
$s set offset for symbol match
$v print ADB variables
$w set output line width

d) calling the shell

! call shell to read rest of line

e) assignment to variables

> name assign dot to variable or register name

Format Summary

a	the value of dot
b	one byte in octal
c	one byte as a character
d	one word in decimal
f	two words in floating point
i	68000 instruction
o	one word in octal
n	print a newline
r	print a blank space
s	a null terminated character string
nt	move to next n space tab
u	one word as unsigned integer
x	hexadecimal
Y	date
^	backup dot
	print string

Expression Summary

a) expression components

decimal integer	e.g. 256
octal integer	e.g. 0277
hexadecimal	e.g. #ff
symbols	e.g. _main main.argc
variables	e.g. <b
registers	e.g. <pc <d0
(expression)	expression grouping

b) dyadic operators

+ add
– subtract
* multiply
% integer division
& bitwise and
| bitwise or
round up to the next multiple

c) monadic operators

~ not
* contents of location
– integer negate

NAME

adb - debugger

SYNOPSIS

adb [-w] [objfil [corfil]]

DESCRIPTION

Adb is a general purpose debugging program. It may be used to examine files and to provide a controlled environment for the execution of UNIX programs.

Objfil is normally an executable program file, preferably containing a symbol table; if not then the symbolic features of adb cannot be used although the file can still be examined. The default for objfil is **a.out**. Corfil is assumed to be a core image file produced after executing objfil; the default for corfil is **core**.

Requests to adb are read from the standard input and responses are to the standard output. If the **-w** flag is present then both objfil and corfil are created if necessary and opened for reading and writing so that files can be modified using adb. Adb ignores QUIT; INTERRUPT causes return to the next adb command.

In general requests to adb are of the form

[address] [, count] [command] [;]

If address is present then dot is set to address. Initially dot is set to 0. For most commands count specifies how many times the command will be executed. The default count is 1. Address and count are expressions.

The interpretation of an address depends on the context it is used in. If a subprocess is being debugged then addresses are interpreted in the usual way in the address space of the subprocess. For further details of address mapping see ADDRESSES.

EXPRESSIONS

The value of dot.

\+ The value of dot incremented by the current increment.

The value of dot decremented by the current increment.

The last address typed.

integer

An octal number if integer begins with a 0; a hexadecimal number if preceded by #; otherwise a decimal number.

integer.fraction
A 32 bit floating point number.

'cccc' The ASCII value of up to 4 characters. \ may be used to escape a '.

< name The value of name, which is either a variable name or a register name. Adb maintains a number of variables (see VARIABLES) named by single letters or digits. If name is a register name then the value of the register is obtained from the system header in corfil. The register names are **d0 ... d7 a0 ... a7 pc ns ac ip**. The register names ns, ac and ip deserve special explanation. The register ns is the concatenation of the 2 byte exception vector number plus the 2 byte 68000 cpustate stored during bus or address error exceptions. If e.g. the first two bytes of ns are 0002, the program was interrupted by a bus error (exception vector 2). The register ac contains the access address for a bus or address error exception. The register ip is the concatenation of the 2 byte instruction register, i.e. the first two bytes of the instruction that caused a bus or address error, and the 2 byte processor status. For exceptions that are not bus or address error, the second half of ns, all of ac, and the first half of ip are 0.

symbol A symbol is a sequence of upper or lower case letters, underscores or digits, not starting with a digit. The value of the symbol is taken from the symbol table in objfil. An initial _ or ~ will be prepended to symbol if needed.

_ symbol
In C, the 'true name' of an external symbol begins with _. It may be necessary to utter this name to distinguish it from internal or hidden variables of a program.

routine.name
The address of the variable name in the specified C routine. Both routine and name are symbols. If name is omitted the value is the address of the most recently activated C stack frame corresponding to routine.

(exp) The value of the expression exp.

Monadic operators

*exp The contents of the location addressed by exp in corfil.

@exp The contents of the location addressed by exp in objfil.

-exp Integer negation.

~exp Bitwise complement.

Dyadic operators are left associative and are less binding than monadic operators.

e1+e2 Integer addition.

e1-e2 Integer subtraction.

e1*e2 Integer multiplication.

e1%e2 Integer division.

e1&e2 Bitwise conjunction.

e1|e2 Bitwise disjunction.

e1#e2 E1 rounded up to the next multiple of e2.

COMMANDS

Most commands consist of a verb followed by a modifier or list of modifiers. The following verbs are available. (The commands '?' and '/' may be followed by '*'; see ADDRESSES for further details.)

?f Locations starting at address in objfil are printed according to the format f.

/f Locations starting at address in corfil are printed according to the format f.

=f The value of address itself is printed in the styles indicated by the format f. (For **i** format '?' is printed for the parts of the instruction that reference subsequent words.)

A format consists of one or more characters that specify a style of printing. Each format character may be preceded by a decimal integer that is a repeat count for the format character. While stepping through a format dot is incremented temporarily by the amount given for each format letter. If no format is given then the last format is used.

The format letters available are as follows.

o 2 Print 2 bytes in octal. All octal numbers output by adb are preceded by 0.
O 4 Print 4 bytes in octal.
q 2 Print in signed octal.
Q 4 Print long signed octal.
d 2 Print in decimal.
D 4 Print long decimal.
x 2 Print 2 bytes in hexadecimal.
X 4 Print 4 bytes in hexadecimal.
u 2 Print as an unsigned decimal number.
U 4 Print long unsigned decimal.
f 4 Print the 32 bit value as a floating point number. Currently only FFP format supported (see fp(3)).
F 8 Print double floating point. For FFP same as above, except larger printing format.
b 1 Print the addressed byte in octal.
c 1 Print the addressed character.
C 1 Print the addressed character using the following escape convention. Character values 000 to 040 are printed as @ followed by the corresponding character in the range 0100 to 0140. The character @ is printed as @@.
s *n* Print the addressed characters until a zero character is reached.
S *n* Print a string using the @ escape convention. *n* is the length of the string including its zero terminator.
Y 4 Print 4 bytes in date format (see ctime(3C)).
i n Print as 68000 instructions. *n* is the number of bytes occupied by the instruction. This style of printing causes variables 1 and 2 to be set to the offset parts of the source and destination respectively.
a 0 Print the value of dot in symbolic form. Symbols are checked to ensure that they have an appropriate type as indicated below.

/ local or global data symbol
? local or global text symbol
= local or global absolute symbol

p 2 Print the addressed value in symbolic form using the same rules for symbol lookup as a.
t 0 When preceded by an integer tabs to the next appropriate tab stop. For example, **8t** moves to the next 8-space tab stop.
r 0 Print a space.
n 0 Print a newline.
"..." 0
Print the enclosed string.

Dot is decremented by the current increment. Nothing is printed.

+ Dot is incremented by 1. Nothing is printed.

- Dot is decremented by 1. Nothing is printed.

newline
If the previous command temporarily incremented dot, make the increment permanent. Repeat the previous command with a count of 1.

[?/]l value mask
Words starting at dot are masked with mask and compared with value until a match is found. If **L** is used then the match is for 4 bytes at a time instead of 2. If no match is found then dot is unchanged; otherwise dot is set to the matched location. If mask is omitted then -1 is used.

[?/]w value ...
Write the 2-byte value into the addressed location. If the command is **W**, write 4 bytes. Odd addresses are not allowed when writing to the subprocess address space.

[?/]m b1 e1 f1[?/]
New values for (b1, e1, f1) are recorded. If less than three expressions are given then the remaining map parameters are left unchanged. If the `?' or `/' is followed by `*' then the second segment (b2,e2,f2) of the mapping is changed. If the list is terminated by `?' or `/' then the file (objfil or corfil respectively) is used for subsequent requests. (So that, for example, `/m?' will cause `/' to refer to objfil.)

>name
Dot is assigned to the variable or register named.

! A shell is called to read the rest of the line following `!'.

$modifier
Miscellaneous commands. The available modifiers are:

- <f Read commands from the file f and return.
- >f Send output to the file f, which is created if it does not exist.
- **r** Print the general registers and the instruction addressed by pc. Dot is set to pc.
- **b** Print all breakpoints and their associated counts and commands.
- **c** C stack backtrace. If address is given then it is taken as the address of the current frame (instead

of a6). If C is used then the names and (32 bit) values of all automatic and static variables are printed for each active function. If count is given then only the first count frames are printed. If the module was compiled with the -L option of cc(1), the procedure name is followed by the current linenumber. For all variables 32 bits are printed, even if the variable is only char or short. In this case only the foremost bytes are to be considered. For register variables, however, the whole register in which the variable resides is printed, and so for shorts the last two bytes are the relevant ones! In order to help you make the distinction between register and normal variables, the latter are prefixed with a "<".

e The names and values of external variables are printed.

w Set the page width for output to address (default 80).

s Set the limit for symbol matches to address (default 255).

o All integers input are regarded as octal.

d Reset integer input as described in EXPRESSIONS.

q Exit from adb.

v Print all non zero variables in octal.

m Print the address map.

:modifier

Manage a subprocess. Available modifiers are:

bc Set breakpoint at address. The breakpoint is executed count-1 times before causing a stop. Each time the breakpoint is encountered the command c is executed. If this command sets dot to zero then the breakpoint causes a stop. If the module has been compiled with option -g, see cc(1), you can set a breakpoint at line 75 by saying L75:b.

d Delete breakpoint at address.

r Run objfil as a subprocess. If address is given explicitly then the program is entered at this point; otherwise the program is entered at its standard entry point. count specifies how many breakpoints are to be ignored before stopping. Arguments to the subprocess may be supplied on the same line as the command. An argument starting with < or > causes the standard input or output to be established for the command. All signals are turned on on entry to the subprocess.

cs The subprocess is continued with signal s, see signal(2). If address is given then the subprocess is continued at this address. If no signal is specified then the signal that caused the subprocess to stop is sent. Breakpoint skipping is the same as for **r**.

ss As for **c** except that the subprocess is single stepped count times. If there is no current subprocess then objfil is run as a subprocess as for **r**. In this case no signal can be sent; the remainder of the line is treated as arguments to the subprocess.

k The current subprocess, if any, is terminated.

VARIABLES

Adb provides a number of variables. Named variables are set initially by adb but are not used subsequently. Numbered variables are reserved for communication as follows.

0 The last value printed.
1 The last offset part of an instruction source.
2 The previous value of variable 1.

On entry the following are set from the system header in the corfil. If corfil does not appear to be a **core** file then these values are set from objfil.

b The base address of the data segment.
d The data segment size.
e The entry point.
m The 'magic' number (0405, 0407, 0410 or 0411).
s The stack segment size.
t The text segment size.

ADDRESSES

The address in a file associated with a written address is determined by a mapping associated with that file. Each mapping is represented by two triples (b1, e1, f1) and (b2, e2, f2) and the file address corresponding to a written address is calculated as follows.

b1<address<e1 => file address=address+f1-b1, otherwise,

b2<address<e2 => file address=address+f2-b2,

otherwise, the requested address is not legal. In some cases (e.g. for programs with separated I and D space) the two segments for a file may overlap. If a ? or / is followed by an * then only the second triple is used.

The initial setting of both mappings is suitable for normal **a.out** and **core** files. If either file is not of the kind expected then, for that file, b1 is set to 0, e1 is set to the maximum file size and f1 is set to 0; in this way the whole file can be examined with no address translation.

So that adb may be used on large files all appropriate values are kept as signed 32 bit integers.

FILES

/dev/mem
/dev/swap
a.out
core

SEE ALSO

ptrace(2), a.out(5), core(5)

DIAGNOSTICS

'Adb' when there is no current command or format. Comments about inaccessible files, syntax errors, abnormal termination of commands, etc. Exit status is 0, unless last command failed or returned nonzero status.

BUGS

A breakpoint set at the entry point is not effective on initial entry to the program.
When single stepping, system calls do not count as an executed instruction.
Local variables whose names are the same as an external variable may foul up the accessing of the external.

EXAMPLES

After a core dump:

```
adb prog          (second parameter core is assumed)
$r                (print registers)
$c                (print C stack trace)
$C                (print full C stack trace)
$e                (print external variables)
bufp/X            (print contents of bufp as long hex)
*bufp/X           (print 4 bytes hex where bufp points to)
write?i           (prints first instruction, link a6,... )
(cr)              (print next instruction)
:b                (set breakpoint at this instruction)
:r hello <inp     (run prog with param hello and input from inp)
main,-1?ia        (disassemble program starting at main)
data/4X2d         (print data: 4 long hex, 2 short decimal)
(cr)              (print next data in same format
#100>d0           (write #100 to register d0)
<d1=X             (print d1 long hex)
*(<a6+8)/XXX      (print first three longs of parameter area)
write?l #4e75     (search for rts instruction at end of proc write)
```

```
:b              (set breakpoint at this address)
:c              (continue program until stop or breakpoint)
:s              (single step through program

etc.
```

On the 68000, as opposed to the 68010 or 68020, the stack will not necessarily grow automatically. You can recognize a stackoverflow with the following commands:

```
adb prog
$r
$m
```

If the value of register ac is less than the value b2 for the second (/) map, the stack overflowed. To get the needed stack size, type

```
#f00000-<ac=D
```

This will output the difference between the top of the stack and the access address in decimal. Round the value up some kbyte, and give the command

```
stksiz prog <value>
```

CADMUS
Testmonitor V1.2

Dokumentations-Nr.: TMV1.2d0384

Best.-Nr.: TMV1.2d0384
Autoren-Kennzeichen: rw

Eingetragene Warenzeichen:

MUNIX	von PCS
DEC, PDP	von DEC
UNIX	von Bell Laboratories

1. Inhalt

CADMUS Testmonitor V1.2 Benutzeranleitung

Rudolf Wildgruber
PCS Gmbh
Pfälzer-Wald-Str. 36
8000 München 90

ABSTRACT

Der *CADMUS Testmonitor V1.2* ist ein Standalone-Programm, das vom *Minitor* von Platte, Floppy-Disk, Band oder Streamer geladen werden kann. Es dient als Steuerprogramm zur Parametrierung und zum Ablauf einzelner Test- und Diagnoseprogramme. Der Testmonitor realisiert eine einheitliche Schnittstelle zu den Test- und Diagnoseprogrammen. Für die Bedienerschnittstelle werden **MUNIX**-Standards verwendet.

Die Version 1.2 unterscheidet sich von Version 1.0 in folgenden Erweiterungen:

- Laden der Testprogramme von Magnetband oder Streamer möglich (sowohl *TM11*- als auch *TS11*-Bänder werden unterstützt)
- Kommandoprozeduren
- Test der Checksumme beim Laden der Testprogramme
- Unterstützung der SLS48/88 - Konsole
- weitere Testprogramme verfügbar

March 23, 1984

CADMUS Testmonitor V1.2 Benutzeranleitung

Rudolf Wildgruber
PCS Gmbh
Pfälzer-Wald-Str. 36
8000 München 90

1. Einführung

Der Testmonitor gliedert sich in 4 Teile:

- Ablaufsteuerung
- Kommando-Interpreter
- Testmonitor-Kommandos
- Systemfunktionen für E/A, Filesystem-Zugriff u.ä.

Die Test- und Diagnoseprogramme sind selbständige Programme, die vom Testmonitor geladen und ausgeführt werden.

1.1. Ablaufsteuerung

- Initialisierungsdialog
- Aufruf des Kommandointerpreters

1.2. Kommando-Interpreter

- Entgegennahme von Benutzereingaben
- syntaktische Korrektheitsprüfung der Benutzereingaben
- Plausibilitätsprüfung von Flag-Kombinationen bei Aufrufen von Testprogrammen
- Ausführung von Kommandos
- Ausführung von Kommandoprozeduren
- Laden von Testprogrammen, Prüfung der Checksumme, Versorgung der Programme mit Parametern und Ausführung

1.3. Testmonitor-Kommandos

- Unterstützung des Benutzers bei Testabläufen
- Zugang zur MUNIX-Dateihierarchie
- einfache Dateifunktionen: Statusabfrage, Listing, Dump

2. Benutzerschnittstelle

2.1. Start

Der Testmonitor wird vom Minitor von einem Datenträger (Platte, Floppy-Disk, Band oder Streamer) geladen. Der Minitor kann von Band oder Streamer nur jeweils die erste Datei (a.out-Format, kein cpio- oder tar-Format) laden. Beim Laden von Platte ist folgendes zu beachten:

> Der Minitor sieht die Filestruktur auf der Platte so, wie sie tatsächlich aufgebaut ist. Daher kann der Dateibaum, wenn die Directory-Struktur geändert wurde - wie z.B. bei der Newcastle-Connection der Fall, - etwas

anders ausehen, wie unter MUNIX. Beim Laden des Testmonitors muss diese Tatsache berücksichtigt werden, und jeweils der vollständige Pfadnamen angegeben werden (z.B. */nodename/*satest).

Mit **g0** wird der Testmonitor wie jedes andere Standalone-Programm gestartet.

Beispiele (Benutzereingaben sind **fett** gedruckt):

Laden des Testmonitors *testmon* (im Directory **/satest**) vom Default-Device:

./satest/testmon
.g0

Laden des Testmonitors *testmon* vom Streamer

.rs
./testmon
.g0

2.2. Initialisierunsdialog

Nach dem Starten des Testmonitors wird ein kleiner Initialisierungsdialog mit dem Benutzer geführt.

Dabei wird gefragt nach dem Eingabemedium und Directory, von dem aus die Testprogramme geladen werden sollen, ob ein Drucker am System angeschlossen ist und nach dem Terminal-Typ.

Nach dem Initialisierungs-Dialog befindet sich der Testmonitor im Kommando-Modus (sichtbar am Prompt '@') und ist bereit Kommandos, Aufrufe von Kommandoprozeduren oder Testprogramm-Aufrufe mit oder ohne Parameter-Angaben zu akzeptieren.

2.2.1. Eingabemedium

Testprogramme und Kommandoprozeduren können von Platte, Magnetband oder Streamer-Kassette geladen werden. Auf Platte wird eine MUNIX-Filesystem-Struktur mit 512 Byte Blockgrösse vorausgesetzt (**Achtung**: Testprogramme können **nicht** von Filesystemen mit **1 kByte** Blockgrösse geladen werden). Auf Band oder Streamer-Kassette müssen die Testprogramme und Kommandoprozeduren im *cpio*-Format abgespeichert sein.

Das Eingabemedium ist in der Form

devname(drive,offset)

zu spezifizieren. Dabei gibt *devname* den Device-Typ an. Zulässige Bezeichnungen für Geräte sind der nachfolgenden Tabelle zu entnehmen. *Drive* ist die Nummer des Laufwerks und die Bedeutung von *offset* ist bei Magnetband und Streamer gegenüber Platte und Floppy-Disk unterschiedlich. Bei Bändern spezifiziert *offset* die Anzahl der Files, die auf dem Band überlesen werden sollen. Bei Platten ist *offset* die logische Blocknummer, ab der gelesen wird.

devname	Device-Typ
rm	RM02/03/05 (80 MB Fujitsu mit DATARAM controller)
rl	RL01/02
hk	RK06/07 (80 MB Fujitsu mit EMULEX controller)
rp	RP03
hp	RP04/05/06, RM02/03 (ohne bad sector handling)
rk	RK05
rx	RX02 Floppy Disk
tm	TM11 Magnetband
ts	TS11 Magnetband
ot	Tandon Winchester und Floppy (OMTI controller)
td	Tandon Winchester (XEBEC controller)
st	SCT11 Streamer

Bei *rx* gibt es eine Besonderheit bezüglich der Laufwerksnummer: *0* und *1* beziehen sich auf single density Laufwerke 0 und 1; *2* und *3* beziehen sich auf double density Laufwerke 0 und 1.

2.2.2. Directory

Nach einem *current directory* wird nur bei Platten als Eingabemedium gefragt. Der Directory-Name ist so anzugeben, wie er der tatsächlichen Dateistruktur auf dem Eingabemedium entspricht, also ohne Mount-Präfixe und eventuell mit dem Knotennamen (bei der Newcastle Connection).

2.2.3. Drucker

Die Frage nach einem angeschlossenen Drucker ist mit **n** (kein Drucker), **s** (serieller Drucker) oder **p** (paralleler Drucker) zu beantworten. Bei einem seriellen Drucker wird zusätzlich nach der Kanalnummer gefragt.
Einige Testprogramme unterstützen nur einen Drucker mit paralleler Schnittstelle. Siehe dazu Abschnitt **BUGS** bei den Kurzbeschreibungen der Testprogramme.

2.2.4. Terminal-Typ

Hier kann der Typ des Konsol-Terminals als String (max. 11 Zeichen) eingegeben werden. Diese Angabe wird von manchen Testprogrammen zur Bildschirmsteuerung ausgewertet. Sinnvolle Eingaben sind *vt100*, *vt52* oder *tvi970*, d.h. nur bei diesen Terminals kann eine entsprechende Bildschirmmaske generiert werden. Bei leerer Eingabe wird *dummy* als Terminal-Typ eingesetzt.
Einige Testprogramme unterstützen nur *VT100*-kompatible Terminals. Siehe dazu Abschnitt **BUGS** bei den Kurzbeschreibungen der Testprogramme.

2.3. Kommandointerpreter

Die im Testmonitor zur Verfügung stehenden Kommandos sind an die Shell-Kommandos von MUNIX angelehnt. Sie werden nachfolgend ausführlicher spezifiziert.

2.3.1. Kommandos

In der Testmonitor-Version V 1.2 stehen folgende Kommandos zur Verfügung:

ls [-l] Inhalt des *current directory* auflisten. Es werden nur die Dateinamen ausgegeben.
Option: **-l** (ausführliches Listing mit Dateigrösse in Bytes und Berechtigungsbits)

cat *file* Ausgabe der Datei *file* auf Konsole

pr *file* Ausgabe der Datei *file* auf Drucker

exit Beenden Testmonitor

help Ausgabe eines Help-Textes auf Konsole

xd *file* Ausgabe der Datei *file* auf Konsole im 'hex-dump'-Format

cd *directory*
Wechsel des *current directory*; jedoch im Gegensatz zu MUNIX nur relativ zum *current directory*. Absolute Directory-Namen werden nicht akzeptiert. *cd* erkennt auch '..' als Directory-Namen, jedoch nur den unmittelbaren Vater; ein Wechsel zu weiter oben liegenden Knoten ist nur durch mehrmaligen Eingabe von 'cd ..' zu erreichen.

cdev *dev* [*dir*]
Wechsel des Eingabemediums mit Angabe eines Directories bei Platten

cterm *type*
Terminal-Typ ändern, z.B. zur Korrektur; als neuer Terminal-Typ wird *type* eingetragen

pwd Ausgabe des *current directory*; es wird jeweils der komplette Pfadnamen mit Device angegeben; die *root* wird dabei mit '.' bezeichnet.

prog [*param-list*]
Aufruf des Testprogramms *prog*:
Laden der Datei *prog* vom Eingabemedium und *current directory*
Prüfung der Checksumme
Versorgung von *prog* mit den angegebenen Parametern
Start von *prog*

proc [*param-list*]
Aufruf der Kommandoprozedur *proc*:
Laden der Datei *proc* vom Eingabemedium und *current directory*
Interpretation des Dateiinhalts als Testmonitor-Kommandos
Angegebene Paramter werden ignoriert. Innerhalb einer Kommandoprozedur können weitere Prozeduren aufgerufen werden. Die maximale Schachtelungstiefe beträgt 10.

2.3.2. Testprogramm-Parameter

Die Testprogramme laufen parametergesteuert ab. Es ist zwischen allgemeinen und speziellen Parametern zu unterscheiden. Allgemeine Parameter sind solche, die für alle oder nahezu alle Testprogramme sinnvoll und anwendbar sind. Spezielle Parameter sind einem einzigen oder sehr wenigen Testprogrammen zugeordnet.

Allgemeine Parameter werden als Teil des Kommandos beim Start eines Testprogrammes angegeben. Der Testmonitor baut einen Parameterblock auf und übergibt diesen an das Testprogramm. Für nicht angegebene Parameter werden Default-Werte verwendet, die abhängig vom Testprogramm sind. Ein allgemeiner Parameter **(dialog)** gibt an, ob für die speziellen Parameter Default-Werte verwendet werden oder ob das Testprogramm mit dem Bediener einen Dialog führen soll.

Allgemeine Parameter sind:

pass Anzahl *no* der Testdurchläufe (0 für Dauertest), wie beim Testprogramm-Aufruf mit *pass=no* angegeben. *0* steht für Dauertest. *1* ist der Default-Wert, wenn beim Aufruf keine Angabe gemacht wurde.

test Liste der Teiltests, die durchgeführt werden.

unit Liste der Units (Testeinheiten: z.B. drives oder controllers), die getestet werden.

flag Angabe von speziellen Flags zum Verhalten des Testprogramms im Fehlerfall. Die Flags sind:

HOE *halt on error*
Testprogramm anhalten und Rückkehr in den Kommandomodus

LOE *loop on error*
In einer unendlichen Schleife den Fehler reproduzieren

IER *inhibit error reports*
Kein Ausdruck einer Fehlermeldung

IXE *inhibit extended error reports*
Kein Ausdruck einer ausführlichen Fehlermeldung, nur kurze Fehlermeldung

PRI *print on line printer*
Ausdruck der Fehlermeldungen auf Drucker

BOE *bell on error*
Akustisches Signal im Fehlerfall

ISR *inhibit statistical reports*
Keine Fehlerstatistik nach einem Testdurchlauf

Die entsprechenden Flags sind gesetzt, wenn sie beim Testprogramm-Aufruf mit *flag=...* angegeben wurden. Per default wird kein Flag gesetzt.

base Basis-Geräteadresse *addr*, die beim Aufruf mit *base=addr* angegeben wurde, sonst *0*.

vec Interrupt-Vektor Adresse *addr*, die beim Aufruf mit *vec=addr* angegeben wurde, sonst *0*.

dialog Das Testprogramm fragt spezielle Parameter im Dialog ab.

s1 Die Zeichenfolge *string* (max. 11 Zeichen), die beim Aufruf mit *s1=string* angegben wurde, sonst leer. Die Bedeutung dieses Parameters legt das Testprogramm fest.

Die allgemeinen Parameter werden beim Aufruf in *param-list* angegeben. Die Syntax für *param-list* ist dabei folgendermassen definiert:

param-list	*param* [*param*]
param	**pass**=*no* | **test**=*no-list* | **unit**=*no-list* | **flag**=*flag-list* | **base**=*no* | **vec**=*no* | **dialog** | **s1**=*string*
no-list	(*no*|*range*) [, (*no*|*range*)]
range	*no*—*no*
flag-list	*flagname* [, *flagname*]
no	eine positive ganze Zahl (einschliesslich 0) oder eine Hexadezimalzahl in der Form **0x***nnnnnn*

flagname	eines der oben spezifizierten Flags
string	eine beliebige Zeichenfolge (maximal 11 Zeichen)

2.3.3. Kommandoprozeduren

Kommandoprozeduren für den automatischen Ablauf von Testprogrammen können nur unter **MUNIX** erstellt werden. Jede Zeile der Kommandoprozedur darf höchstens ein Testmonitor-Kommando, einen Testprogramm-Aufruf oder einen Kommandoprozedur-Aufruf enthalten. Leerzeilen sind erlaubt, Kommentare verboten.
Die Abarbeitung einer Zeile der Kommandoprozedur wird protokolliert.

2.4. Terminalhandling

Für die Terminalein- und ausgabe stehen nachfolgend beschriebene Steuerzeichen zur Verfügung:

CTRL S: Anhalten einer gerade laufenden Terminal-Ausgabe.

CTRL Q: Fortsetzen einer mit *CTRL S* angehaltenen Terminal-Ausgabe.

CTRL X: Löschen der aktuellen Eingabezeile im Kommandomodus.

CTRL C: Unterbrechen eines gerade laufenden Programmes.

3. Beispiele

Anhand des Testprogrammes *check* soll ein kompletter Dialog durchgespielt werden:

Dabei werden die Ausgaben des Testmonitors **fett** gedruckt dargestellt, sowie die Eingaben des Benutzers in *Kursiv-* Schrift.

Es wird angenommen, dass sowohl der Testmonitor wie auch die Testprogramme bereits unter **MUNIX** in das Directory **/satest** im Filesystem */dev/hk0* eingespielt worden sind, und die Directory-Struktur unverändert (also keine Superroot der Newcastle Connection) vorliegt.

Nach dem Einschalten des Systems bzw dem Betätigen des INIT-Schalters erscheint an der Konsole

MINITOR

Mit der Eingabe

/satest/testmon

wird der Testmonitor von Platte (80 MB Fujitsu, Laufwerk 0) geladen und es meldet sich wieder der Minitor mit dem Prompt '.'
Mit

g0

wird der Testmonitor nun gestartet. Er meldet sich mit dem Text:

CADMUS-TESTMONITOR V1.2

input-device xx(d,o):

Nach Eingabe des Eingabemediums, (z.B. rx(2,0) für das erste Floppylaufwerk) oder im Beispiel für die Platte

hk(0,0)

fragt der Testmonitor weiter:

directory

Die Eingabe

/satest

stellt das *current directory* ein.
Der weitere Dialog:

line printer (n/s/p): *s*

line number (1-7): *1*

terminal : *vt100*

@

Nun ist der Testmonitor bereit, Kommandos oder Programmnamen zu akzeptieren, im Beispiel:

check s1=hk-w unit=1 flag=PRI,IXE

Alle weiteren Ausgaben erscheinen am Drucker, bis sich der Testmonitor wieder mit @ zurückmeldet.

Ein weiteres Beispiel ist der Aufruf des SCSI-Testprogramms **scsitst**:

Das Kommando

@scsitst pass=3 test=1,3-6 unit=0,1 s1=d503

führt zu folgendem Testablauf:

Die Tests mit den Nummern 1,3,4,5 und 6 werden in dieser Reihenfolge ausgeführt. Dieser Testdurchlauf erfolgt insgesamt dreimal. Getestet werden unit 0 und unit 1, d.h. 2 Winchester-Drives. Die Winchester sind vom Typ TM 503. Es wird der Controller DTC 520 verwendet.

4. I/O Page Adressen

Für Konsole und seriellen Drucker wird sowohl eine DLV11 als auch eine SLS48/88-Schnittstelle unterstützt.

In der Version 1.2 sind die Adressen und Interruptvektoren von Konsole und Drucker fest eingestellt und können auch nicht durch die Parameter *base* und *vec* verändert werden.

Die in V1.2 gültigen Adressen und Interrupt-Vektoren sind nachfolgend angegeben. Beachten Sie bitte:

- die oktal angegebenen Adressen sind direkt auf die Controller-Switches zu übertragen;
- die sedezimal angegebenen Adressen in Klammern stellen die echten Adressen im 68000-Adressraum dar und sind bei den Interruptvektoren das Vierfache der oktal angegebenen Adresse.

Gerät	Interruptvektor		I/O-Page Adresse	
	oktal	hex	oktal	hex
DLV11-Konsole	60	C0	777560	FFFF70
paralleler Drucker	200	200	777514	FFFF4C
serieller Drucker:				
DLV11-line 1	300	300	776500	FFFD40
DLV11-line 2	310	320	776510	FFFD48
DLV11-line 3	320	340	776520	FFFD50
DLV11-line 4	330	360	776530	FFFD58
DLV11-line 5	340	380	776540	FFFD60
DLV11-line 6	350	3A0	776550	FFFD68
DLV11-line 7	360	3C0	776560	FFFD70
SLS48/88	300	300	776000	FFFC00

5. Verfügbare Testprogramme und Kommandoprozeduren

5.1. Testprogramme

In V1.2 sind nachfolgend aufgeführte Testprogramme verfügbar:

bsctest	BSC-KE Testprogramm
check	Platten-Test- und Formatierprogramm
dztst	DZV11 Testprogramm
fpptst	Floating Point Processor Test
lbptst	CANON Laser Beam Printer Test
memtst	Speichertest
muxketst	MUX-KE Testprogramm
qetest	3COM Q-Bus Ethernet Controller Testprogramm
scsitst	SCSI-Adapter Test
slutst	SLS48/SLS88 Test
t68030	QU68030/50 Prozessortest (ohne MMU-Stufe 2)
t68050	QU68050 Prozessortest (nur MMU-Stufe 2)
tdma	DMA Testprogramm

Dabei spielen die Programme ***memtst*** und ***lbptest*** eine gesonderte Rolle, da diese nach Ablauf nicht mehr in den Testmonitor zurückkommen. Der Testmonitor muss danach mit dem Minitor neu geladen und gestartet werden.
Für *qetest* wird ein MUNIX-Programm (kein Standalone-Programm) *echoserver* mitgeliefert, das von einem anderen *CADMUS*-System mit *qetest* kommuniziert.

5.2. Kommandoprozeduren

In V1.2 werden standardmässig 4 Kommandoprozeduren ausgeliefert *(emuinit, emuinitpr, emuinit2, emuinit2pr)*. Sie dienen zur Initialisierung (Formatierung und Bad Sector Test) von Platten, die an den EMULEX SC02 Controller (RK06/07-Emulation) angeschlossen werden.

6. Installation

Der Testmonitor ist mit den Testprogrammen Bestandteil der MUNIX-Basissoftware (ab V1.5). Im MUNIX-Dateibaum ist er im Directory **/satest** oder **/usr/satest** zu finden.

Separat erfolgt die Lieferung des Testmonitors V1.2 mit den erwähnten Programmen auf Band, Streamer-Kassette oder Floppy-Disk:

Die Floppy ist im MUNIX-Filesystem-Format beschrieben (512 Byte Blockgrösse). Das Magnetband bzw. die Streamer-Kassette enthält als erste Datei den Testmonitor im a.out-Format und als zweite Datei die Testprogramme und Kommandoprozeduren im *cpio*-Format. Der Testmonitor kann also direkt von diesen Datenträgern geladen werden. Um die Testprogramme vom Testmonitor zu laden, ist dann als Eingabemedium *rx(2,0)*, *tm(0,1)* oder *st(0,1)* einzugeben.

Wir empfehlen, die Programme auch unter die Root einzulesen, damit sie direkt von der Winchester geladen werden können. Folgende Kommandos sind z.B. notwendig, um die Streamer-Kassette einzulesen:

```
cd /
mkdir satest
cd satest
stskip 1
cpio -ivmS < /dev/nrst0   (Standalone-Testprogramme)
cd /
cpio -ivmS < /dev/rst0    (Echoserver)
```

NAME

bsctest - KE-BSC1/2 (KE-SI01 + KE-V24 / KE-SI02) - test

SYNOPSIS

bsctest [param-list]

DESCRIPTION

Bsctest checks 4 (2) synchronous BSC channels in short circuit mode (short circuit plug K907.027:

channel 0 < - > channel 1
channel 2 < - > channel 3).

Data and control lines are checked in both directions.

Hardware:	KE-BSC2 +KE-SI02	or	KE-BSC1 +KE-SI01 [KE-SI01] +KE-V24 [KE-V24]	or	CP-WCS +KE-SI02
(default:	KE-BSC2 +KE-SI02)				

The jumper configuration for the boards is printed on the console if *bsctest* is started with the parameter **dialog** .

PARAM-LIST

pass Number of test passes or **0** for infinite test loop. The default value is **1** .

flag **HOE** Halt on error and return to testmonitor.
BOE Bell on error.
LOE Loop on error. Number of test passes is set to **0** (infinite test loop).

unit Number of channels (**2** or **4**). Default is **4** .

dialog The test program asks interactively for test commands. Jumper configurations are printed according to hardware configurations.

EXAMPLES

Test with 10 passes, halt on error and bell on error, 2 channels

bsctest pass=10 flag=HOE,BOE unit=2

FILES

KEBSC2.KEA
KEBSC2.MAP

SEE ALSO

T920.518 MUNIX RJE mit BSC-KE-Controller
T922.103/104/105.01 KE-SI02 Testvorschrift

NAME

check – disk checking and formatting

SYNOPSIS

check s1=devname[-(**w**|**u**)] [param-list]

DESCRIPTION

Check is the *CADMUS* disk checking program. Additionally it has a formatting capability for disks with standard headers. 5 1/4" winchesters emulated by the *Andromeda WDC11* controller as RL02 can also be formatted. As disk checking is done without interrupts a separate test for controller interrupts was added.

The main purpose of *check* is to test disks for the location of bad sectors and to write the *bad sector file* onto disks. The bad sector file is a list of all bad sectors found on a disk. *MUNIX* uses this information to avoid allocating bad sectors to a user's file. If there is an error in a header, or if there is a read or write error within one sector, that sector is defined as a *bad sector*. If possible the header of this sector is marked.

Check is aborted if one of the following conditions occurs:

- controller not accessible at specified or default address
- bad status on all specified or default units
- too many bad blocks (more than 126) on a tested unit

If there is a bad status on one unit, that unit is omitted from testing.

A brief explanation of *check* is available in *check.help* Use the testmonitor *cat* command to get it on screen.

PARAM-LIST

s1 The device name (see table below) of the disks to be tested optionally followed by **–w** or **–u** .
The devices in the following table are supported by *check*. The listed CSR and vector addresses (hex notation) are default values. They can be changed by the **base** resp. **vec** parameter. Devices indicated by *STD* in the column *Formatting* uses standard headers. Those indicated by *WDC11* need the *ANDROMEDA WDC11* controller to be formatted.

Supported Devices

Device Name	Disk Type	CSR Address	Vector Address	Formatting
hk	RK06/07	FFFF20	220	STD
rl	RL01/02	FFF900	1C0	WDC11
hl	RL01/02	FFF910	1D0	WDC11
rm	RM02/03/05	FFFDC0	2B0	STD

The option **-w** opens the disk in *read/write-mode*, while **-u** opens the disk in *update-mode*. A missing option opens the disk in *read-only-mode*.

In *read/write-mode* the contents of the disk is overwritten, bad sectors are marked, the bad sector file is initialized or modified

and formatting is possible. This is the proper mode for new disks.
In *update-mode* sectors ar tested only by reading, bad sectors are marked and the bad sector file is initialized or modified. Formatting is inhibited.
In *read-only-mode* the contents of the disk is left unchanged. Sectors are tested only by reading, no formatting is done, no sector is marked as bad and the contents of an existing bad sector file is not modified.
For example (user input is **bold**):

s1=hk-w or
s1=rm-u or
s1=rl

pass Number of test passes or **0** for infinite test loop. The default value is **1**

test A list of testnumbers in the range *0* to *6* If no test-list is specified, tests *0*, *1*, *2* and *6* are executed by default. During a test pass the tests are executed in increasing order. The test descriptions refer to disks opened in *read/write-mode*. If you opened a disk in another mode read the descriptions accordingly:

0 *Bad Sector Scan:*
The complete disk is tested. Sectors are first written in increasing order and then read in decreasing order. A bad sector file is written onto the disk.

1 *Sector Range Test:*
A range of consecutive sectors is tested. By default these are the sectors *0* to *1999* . If the parameter **dialog** is specified you are interactively asked for the starting sector and the number of sectors to be tested. The sectors are tested alternately: *1st sector*, *last sector*, *2nd sector*, ... to the midst of the given interval. The already existing bad sector file is updated.

2 *Random Sector Test:*
Test disk sectors in random order. By default *2500* sectors are tested. If the parameter **dialog** is specified you are interactively asked for the number of sectors to be tested. The already existing bad sector file is updated.

3 *Inspect Bad Sector File:*
List the contents of an existing bad sector file.

4 *Append Bad Sectors:*
Bad sectors are appended manually to an existing bad sector file. When the message *enter sector numbers (-1 to end)* appears on the screen, type in the numbers of sectors to be marked as bad sectors. To exit from this test enter **−1** .
This test is extremely helpful if you know any bad sectors not detected by the check program.

5 *Format Disk:*
Write good sector headers and initialize data fields optionally on the complete disk volume or on single tracks. If the parameter **dialog** is not specified the complete disk is formatted. If **dialog** is specified you are interactively asked for complete or single track formatting.

6 *Controller Interrupt Test:*
A *RESET* or *DRIVE CLEAR* command is executed with interrupt enabled. The test is successful if the controller interrupts within a certain time (about 1 second). There are three kinds of possible error conditions:

– *timeout:* no interrupt occured
– *spurious interrupt:* daisy chain not closed
– *illegal interrupt:* wrong interrupt address

As a proper test strategy for new disks we suggest first to format all drives with test **5** and then to start the default tests for all drives.
If there is a new bad sector on an already used disk test a small range around the bad sector by test **1** (**dialog** specified) or use test **4** to mark the bad sector manually. Used disks should be checked in *read-only-mode*.

unit A list of units (disk drive numbers) to be tested. *0* is default. For *hk* and *rm* a disk drive number is in the range *0* to *7* , for *rl* and *hl* the range is *0* to *3*

flag A list of flags (see CADMUS Testmonitor Benutzeranleitung). **PRI** makes a line printer protocol. **IXE** and **IER** suppress detailed error messages. **HOE** terminates *check* immediately if there is a bad status on one unit or if the controller interrupt test fails. **BOE** bells on an error condition. All other flags are ignored

base A nonstandard CSR address

vec A nonstandard interrupt vector address.

dialog Ask interactively for special parameters. Applies to tests *1*, *2* and *5*

EXAMPLES

Formatting a complete 80 MB Fujitsu winchester (EMULEX controller):
check s1=hk-w unit=0-2 test=5
Default tests with line printer protocol:
check s1=hk-w unit=0-2 flag=PRI
Formatting a complete 80 MB Fujitsu winchester (DATARAM controller):
check s1=rm-w test=5
Testing a user defined sector range in read-only-mode
check s1=rl test=1 unit=1 dialog flag=PRI

FILES

check.help

SEE ALSO

rl(4), rm(4), hk(4), iopage(7)
Bad Sector Handling
CADMUS Testmonitor Benutzeranleitung

BUGS

The interrupt test doesn't work with *DATARAM S04/A* controller. Ignore the *timeout* message.

NAME

dztst – DZV11 test (without send interrupts)

SYNOPSIS

dztst [param-list]

DESCRIPTION

Dztst is the *CADMUS* DZV11 test program.

Dztst tests simultanously two DZV11 channels. One acts as the transmitter the other as the receiver. In this way all DZV11 channels can be tested.
In test 1 data is only transmitted, while test 2 transmits and receives data using two channels. Test 3 checks the signals: DTR,RING LINE and CARRIER LINE. For the tests 2,3 you need a short circuit plug, connecting the channels *0* and *1* , *2* and *3* etc.

PARAM-LIST

pass Number of test passes or **0** for infinite test loop. The default value is **1** .

test A list of testnumbers in the range *1* to *3* If no test-list is specified, tests *1* to *3* are executed by default. During a test pass the tests are executed in increasing order.

1 *Transmit test:*
The output channel is tested (255 characters will be transmitted).

2 *Transmit - receive test:*
The output and the input channels will be tested (255 characters are transmitted from output to input channel).

3 *Modem test:*
The signals: DTR, CARRIER LINE and RING LINE will be tested.

unit Input and output channels. All are default.

flag A list of flags (see CADMUS Testmonitor Benutzeranleitung). **PRI** makes a line printer protocol. **IER** suppresses error messages. **BOE** bells on error. **HOE** halts on error. **LOE** loops on error.

base This parameter is ignored.

vec This parameter is ignored.

dialog If specified you are interactively asked for the number of characters to be transferred, the channels to be tested and display flags.

EXAMPLES

Default tests (1,2,3); Channels: All (0<>1,2<>3,..) ; 255 characters.
dztst

Default tests with dialog
dztst dialog

Only test 1; output channel 2 ; 255 characters.

dztst test=1 unit=2

Only test 2; channel 3 and 5; loop on error; bell on error

dztst test=2 unit=3,5 flag=LOE,BOE

SEE ALSO

Beschreibung fuer DZV11 - Testprogramm
CADMUS Testmonitor Benutzeranleitung

NAME

emuinit – format and check an 84 MB Fujitsu drive with EMULEX SC02 controller

SYNOPSIS

emuinit
emuinitpr
emuinit2
emuinit2pr

DESCRIPTION

These command procedures format and check complete 84 MB Fujitsu drives with EMULEX SC02 controller. They can be used for the initial set-up of winchester drives.
Emuinit and *emuinitpr* refer to the first drive, while *emuinit2* and *emuinit2pr* refer to the second drive. *Emuinitpr* and *emuinit2pr* make a line printer protocol, while *emuinit* and *emuinit2* only print a protocol to the console.

PARAM-LIST

All params are ignored.

SEE ALSO

check(TM)
Bad Sector Handling
CADMUS Testmonitor Benutzeranleitung

NAME

fpptst – test program for FPP 81 (floating point processor)

SYNOPSIS

fpptst [param-list]

DESCRIPTION

Fpptst serves as a GO/NOGO test program for the floating point processor FPP 81. Both single loop tests and infinite loop tests are possible. For a detailed description of **fpptst** see *FPP 81 Floating Point Prozessor Testanweisung*.

PARAM-LIST

pass Number of test passes or **0** for infinite test loop. The default value is **1** .

test A list of testnumbers in the range *1* to *30*. If no test-list is specified, all tests are executed by default. During a test pass the tests are executed in increasing order.

1..19, 21, 22
Instruction tests: all floating point instructions are tested (single precision only) with different address modes.

20 *Compare (CMP) / CSR-TEST*

23..27 *Address mode tests*

28 *Trap test*

29 *Double precision test*

30 *Op. buffer - result buffer - transfer test*

unit This parameter is ignored.

flag A list of flags (see CADMUS Testmonitor Benutzeranleitung). **PRI** makes a line printer protocoll. **IER** suppresses error messages. **BOE** bells on error. **HOE** halts on error. **LOE** loops on error.

base This parameter is ignored.

vec This parameter is ignored.

sl This parameter is ignored.

dialog The test program asks interactively for test commands.

SEE ALSO

FPP 81 Floating Point Prozessor Testanweisung
CADMUS Testmonitor Benutzeranleitung

NAME

lbptst – laser beam printer test

SYNOPSIS

lbptst

DESCRIPTION

Lbptst is the test program for the CANON laser beam printer. *Lbptst* overwrites the testmonitor in main memory. After termination of *lbptst* the testmonitor has to be loaded from disk, tape or streamer by the *Minitor*.

PARAM-LIST

All params are ignored. *Lbptst* asks the user for test commands.

SEE ALSO

Minitor-Manual
CADMUS Testmonitor Benutzeranleitung
LBP68000 LASER PRINTER SYSTEM - Testanweisung

BUGS

The console is dead when the *LBP* controller is missing. Push INIT.

NAME

memtst – memory test

SYNOPSIS

memtst

DESCRIPTION

Memtst is the *CADMUS* memory test program. *Memtst* overwrites the testmonitor in main memory. After termination of *memtst* the testmonitor has to be loaded from disk, tape or streamer by the *Minitor*.

PARAM-LIST

All params are ignored. *Memtst* asks the user for test commands.

SEE ALSO

Minitor-Manual
CADMUS Testmonitor Benutzeranleitung
Beschreibung fuer Speichertestprogramm MEMTST

BUGS

Memtst supports only a parallel line printer.

NAME

muxketst – MUX-KE test (without interrupt handling)

SYNOPSIS

muxketst [param-list]

DESCRIPTION

Muxketst is the *CADMUS* MUX-KE test program.

Muxketst tests simultanously two MUX-KE channels (DH). One acts as the transmitter the other as the receiver.
In test 1 data is only transmitted, while test 2 transmits and receives data using two channels. For test 2 you need a short circuit plug.

PARAM-LIST

pass Number of test passes or 0 for infinite test loop. The default value is 1 .

test A list of testnumbers in the range *1* to *2* If no test-list is specified, tests *1* and *2* are executed by default. During a test pass the tests are executed in increasing order.

1 *Transmit test:*
The output channel is tested (255 characters will be transmitted).

2 *Transmit - receive test:*
The output and the input channels will be tested (255 characters are transmitted from output to input channel).

unit Input and output channel (max. 2 channels). *0* and *1* are default.

flag A list of flags (see CADMUS Testmonitor Benutzeranleitung). **PRI** makes a line printer protocol. **IER** suppresses error messages. **BOE** bells on error. **HOE** halts on error. **LOE** loops on error.

base This parameter is ignored.

vec This parameter is ignored.

dialog If specified you are interactively asked for the number of characters to be transferred, the channels to be tested and display flags.

EXAMPLES

Default tests (1,2); Channels: 0 and 1 ; 255 characters.
muxketst

Default tests with dialog
muxketst dialog

Only test 1; output channel 2; 255 characters.
muxketst test=1 unit=2

Only test 2; channel 3 and 5; loop on error; bell on error
muxketst test=2 unit=3,5 flag=LOE,BOE

SEE ALSO

Beschreibung fuer MUX-KE - Testprogramm
CADMUS Testmonitor Benutzeranleitung

NAME

qetest - test 3 Com QBus Ethernet-Controller QE

SYNOPSIS

qetest [param-list]

DESCRIPTION

qetest is a check and diagnostic program for the 3 Com Ethernet Controller QE. **qetest** offers several functional tests for the memory, control and serial board of the controller.
Memory and Transmit-functions are testable (with some restrictions) without another Ethernet-Controller. The Receive-function-test and test of data-integrity at transmission need an echoserver running on a remote unix-machine. The echoserver should reside in */usr/bin*.
qetest is aborted if control-registers or buffermemory-addresses are not accessible. There is no checking of the interrupt-vector because qetest runs in polling-mode.
After every pass qetest may be forced to detail error-messages.

PARAM-LIST

dialog In dialog-mode qetest gives detail information before running a test.

pass Number of test passes; 0 for infinite test Coop. The default value is 1. An infinite test loop can be stopped with CNTR-C.

test A list of testnumbers in the range of 0 to 4. If no test-list is specified, test 0, 1, 4 are executed by default. Tests are executed in increasing order during a test pass.

0 Test of buffer memory: Addressing, Data and byte-operations are tested.

1 Transmit a single packet using each buffer. The test checks transmit-done, jam-occurrence and buffer-release. No echoserver is required.

2 Transmit multiple packets. All 16 buffers are prepared for transmission, then the controller is started. Checking is done analogous test 1.

3 Receive a single packet using each buffer. Each packet on the Ethernet is received independent of its destination address. (Monitoring Ethernet cable).

4 Echo-Test: Transmit and Receive of a single packet with check of data integrity.

An echoserver on a remote unix machine is necessary to reflect the Ethernet packet. If test 4 is selected, a dialog asks the addressing parameters before starting the test-passes.

flag HOE halt on error.
LOE loop on error. qetest loops the faulty test independently of further errors until break.
IER, IXE inhibit error reports. Only the error-count of the actual pass and the total error-count are displayed. After every error report a detailled explanation of the error-codes may be asked by the user.

EXAMPLE

Infinite echo-test:

```
qetest test=4  pass=0
```

FILES

/usr/bin/echoserver

SEE ALSO

Ethernet-Controller 3C200
Installation und Standalone-Test
im System QU68000.
T923.408.01

NAME

scsitst– test program for SCSI adapter.

SYNOPSIS

scsitst [param-list]

DESCRIPTION

Scsitst Version 3.0 tests the SCSI-Bus-Controller and the SCSI-Adapter with a 5 1/4" Winchester or a 5 1/4" Floppy.

PARAM-LIST

pass Number of test passes or **0** for infinite test loop. The default value is **1**

test A list of testnumbers in the range *0* to *6* . If no test-list is specified, test *2 (read sequent)* is executed by default.

- **0** *Read block* - you are asked for the number of the block to be read.
- **1** *Write block* - you are asked for the number of the block to be written.
- **2** *Read sequent* - read the blocks of the data medium (floppy or winchester) sequentially.
- **3** *Write sequent* - write blocks sequentially.
- **4** *Write + read + compare sequent*
- **5** *Write + read + compare zigzag*
- **6** *Write sequent read + write random*

unit Logical drive number (**0..3**). Unit **0** winchester 0, unit **1** winchester 1, unit **2** floppy 0 and unit **3** floppy 1.

s1 Controller- and drive type. **D** or **d** for DTC 520 controller. **O** or **o** for OMTI 20D controller. **503** for TM 503 winchester, **603** for TM 603 winchester, **703** for TM 703 winchester, **100-4** for TM 100-4 or TM 101 floppy, **208** for RO 208 winchester, **1065** for Maxtor 1065 winchester and **6185** for BASF 6185 winchester. For example **D503** for DTC 520 controller and TM 503 winchester.

dialog The test program asks interactively for test commands.

EXAMPLES

Zigzag-test, 1 pass, with DTC 520 controller and TM 503 winchester as unit 1.

scsitst pass=1 unit=1 s1=D503 test=5 oder
scsitst pass=1 unit=1 s1=d503 test=5

Interactive testing.

scsitst dialog

SEE ALSO

Beschreibung fuer SCSI - Testprogramm
CADMUS Testmonitor Benutzeranleitung

BUGS

Scsitst works fine only with a VT100-compatible terminal.

NAME

slutst – SLS-48 and SLS-88 test program.

SYNOPSIS

slutst [param-list]

DESCRIPTION

Slutst Version 3.0 tests the serial lines of the Multi-Function-Board (MFB). To run this test you need a *DLV11* for the console interface (standard *DEC*-console).

PARAM-LIST

pass Number of test passes or **0** for infinite test loop. The default value is **1** .

dialog The test program asks interactively for test commands.

EXAMPLES

Short circuit test with 100 passes

slutst pass=100

Interactive testing.

slutst dialog

SEE ALSO

SLS 88 - Testprogramm
CADMUS Testmonitor Benutzeranleitung

BUGS

Slutst works fine only with a VT100-compatible terminal.

NAME

t68030 – processor test for QU68030/50

SYNOPSIS

t68030 [param-list]

DESCRIPTION

T68030 version 3.1 tests the QU68030/50. The program checks for MC68000 or MC68010. The second mmu-stage of QU68050 is not tested (see *T68050(TM)*).

PARAM-LIST

pass Number of test passes or **0** for infinite test loop. The default value is **1** .

test A list of testnumbers in the range *0* to *10*. If no test-list is specified, tests *0* *7* and *10* are executed by default. During a test pass the tests are executed in increasing order.

0 (ed) test of basic processor functions

1 (ad) test of address modes.

2 (be1) test of 1-operand instructions.

3 (be2) test of 2-operand instructions.

4 (rr) RAM-ROM test.

5 (mt) MMU - test.

6 (sd) test of MMU - segment descriptors

7 (ir) interrupt test.

8 is ignored. This test (force parity error) can only be started interactively.

9 (pf) Page-Fault-Test. (Nur QU68050)

10 (bt) bus timeout test

flag HOE Halts (program termination) on error and returns to test-monitor. PRI prints error messages to line printer. IER suppresses extended error messages.

dialog The test program asks interactively for test commands.

EXAMPLES

Test with extended error messages, do not halt on error.

t68030

100 test passes, halt on error and suppress extended error messages.

t68030 pass=100 flag=HOE,IER

Interactive testing.

t68030 dialog

SEE ALSO

Programmbeschreibung t68030
CADMUS Testmonitor Benutzeranleitung

BUGS

T68030 works fine only with a VT100-compatible terminal. Only a parallel line printer is supported.

NAME

t68050 – MMU 2 test program

SYNOPSIS

t68050 [param-list]

DESCRIPTION

T68050 Version 3.0 tests the second MMU-stage of the QU68050.

PARAM-LIST

pass Number of test passes or **0** for infinite test loop. The default value is **1** .

EXAMPLES

Test with 100 passes.

t68050 pass=100

SEE ALSO

Kurzbeschreibung fuer MMU2 - Testprogramm
CADMUS Testmonitor Benutzeranleitung

BUGS

T68050 works fine only with a VT100-compatible terminal.

NAME

tdma – DMA - test program

SYNOPSIS

tdma [param-list]

DESCRIPTION

Tdma version 3.0 is running under QU68030 or QU68050. For this test you have to plug a test-CP to the CADMUS-System.
The test-CP is a *Communication Processor* (in German *Kommunikations-Element* or *KE)* with special firmware (test proms). If you want to order the test-CP please contact *PCS.*

PARAM-LIST

pass Number of test passes or **0** for infinite test loop. The default value is **1** .

test A list of testnumbers in the range *0* to *2.* If no test-list is specified all tests are executed by default. During a test pass the tests are executed in increasing order.

0 *DMA-read* - read blocks of data from memory.

1 *DMA-write* - write blocks of data to memory.

2 *DMA-move* - read blocks of data from memory and write to memory.

unit Number of the associated DMA extension register (**0, 2, 3**). Default = 0 (DER0).

s1 DMA request time in ms (**5..255**). Default = 5 ms.

dialog The test program asks interactively for test commands.

EXAMPLES

Test with 100 passes, DMA extension register 2, DMA request time 255 ms.

tdma pass=0 unit=2 s1=255

Interactive testing

tdma dialog

SEE ALSO

Kurzbeschreibung fur DMA-Testprogramm des QU68030/50
CADMUS Testmonitor Benutzeranleitung

BUGS

Tdma works fine only with a VT100-compatible terminal.

Beschreibung
für
Speichertestprogramm MEMTST

Martin Gröber

Periphere Computer Systeme
Pfälzerwald Str. 36
D-8000 München 90

1. Einleitung

Das Testprogramm "MEMTST" ist ein Speichertestprogramm, das in 68000-Assembler-Sprache geschrieben ist. Mit Hilfe dieses Programmes kann ein beliebig grosser Q-Bus- und/oder S-Bus-Speicher ausgetestet werden. "MEMTST" ist speziell auf den Prozessor QU68000 (QU68030/50) zugeschnitten. (Das Programm nutzt das lokale RAM des QU68000.)

2. Voraussetzung

2.1. Hardware

QU 68000 System
QU 68000 Prozessor
Serielle Schnittstelle
FLoppyDisk oder anderer Massenspeicher
Terminator

Option: parallele Drucker-Schnittstelle mit Drucker

2.2. Software

CADMUS Testmonitor
Testprogramm MEMTST

"MEMTST" ist ein Standalone-Testprogramm, das ohne Betriebssystem läuft. Gestartet wird "MEMTST" vom Testmonitor. Optional kann "MEMTST" auch direkt vom Minitor geladen werden.

3. Bedienerführung

3.1. Laden und Starten des Testprogrammes

Durch Angabe des Programmnamens

memtst

wird "MEMTST" vom Testmonitor geladen. Ohne Testmonitor kann "MEMTST" auch direkt vom "Minitor" mit folgenden Befehlen geladen werden.

Eingabe:

rx<cr>	Laden von Floppy
/memtst<cr>	
g0<cr>	

Nach jedem <cr> meldet sich der "Minitor" mit dem Prompt. Durch g0 wird das Programm gestartet.
Memtst meldet sich mit folgender Systemausgabe.

```
Q U 6 8 0 0 0   M E M O R Y   T E S T  Vers. x.x

Prozessor - Version : MC 680xx
-----------------------------------!------

***** Lokaler RAM-Bereich o.k. *****

Gesamt-Speicherkapazitaet 00xxxx KByte  o.k. (CR/n) ?
```

Das Programm stellt selbststaendig die Grösse des Speichers fest. Die Grösse wird durch obigen Ausdruck angezeigt und muss durch Eingabe von "j" oder <cr> bestätigt werden.

```
Testprogramm-Auswahlliste:
------------------------------

Housenumber   = 0   Read/Modify/Write   = 5
Random        = 1   Write Byte          = 6
Bit-Shifting  = 2   Write Long Word     = 7
Refresh       = 3   Parity              = 8
Opcode        = 4   Force Parity Error  = 9
                    Tests 0....8        = CR


Testnummer (getrennt d. ',') :
```

Das Programm bietet mehrere Testmodule zur Auswahl an. Beschreibung der Testmodule siehe Programmbeschreibung. Nach dem Ausdruck der Systemmeldung hat das Programm den lokalen Speicherbereich des QU68000 bereits getestet. Tritt bei diesem Test ein Fehler auf, wird dies durch eine entsprechende Fehlermeldung angezeigt. Der lokale RAM-Bereich des QU68000 wird für den weiteren Programmablauf benutzt.

3.2. Programmeingaben

Testnummer (getrennt d. ',') :

Hier koennen die verschieden Tests ausgewaehlt werden, die ausgefuehrt werden sollen. Es koennen auch mehrere Tests ausgewaehlt werden. Die einzelnen Tests sind durch "," zu trennen. Wird nur <CR> eingegeben, werden die

Tests 0-8 ausgefuehrt.
Fuer den Test 8 (Parity) muss der Speicher entsprechend den Standardeinstellungen fuer den QU68000 verdrahtet sein. D.h. Wort-Parity-Uebertragung auf Leitung BDAL16. (Kein CSR)
Der Test 9 (Force Parity Error) ist nicht mit allen Speichern moeglich. Der Speicher muss die Option: "Write wrong Parity" besitzen.

Speicher-Teilbereich testen (j,CR) ?

Soll nur ein Teilbereich des Speichers getestet werden muss hier "j" eingegeben werden. Mit "n" oder <cr> wird der gesamte erkannte Speicher getestet.
Bei einem Prozessor mit S-Bus-Speicher kann das Speichertestprogramm ueber den S-Bus oder ueber den Q-Bus ablaufen. Die Auswahl wird mit der folgenden Abfrage getroffen.

S-Bus abschalten (j,CR) ?

Dauertest ?

Es besteht die Möglichkeit, das Programm im Dauertest zu betreiben.

3.3. Testprotokoll auf Drucker ?

Das Testprotokoll kann zusaetzlich zur Terminal-Ausgabe auf einem Drucker ausgegeben werden. Bei Dauertest werden nur die Fehlermeldungen ausgegeben. Wird eine Anzahl von 20 Druckseiten ueberschritten, wird die Druckerausgabe automatisch beendet.

3.4. Programm-Abbruch

Das Programm kann jederzeit durch die Eingabe von "Contr C" abgebrochen werden.

3.5. Fehler- und Statusreport

Waehrend des Programmlaufes wird immer angezeigt, welcher Teiltest gerade laeuft. Durch Druecken einer beliebigen Taste (ausser Contr C) wird eine Statuszeile ausgegeben. Die Ausgabe enthaelt: Test-Nr., Adresse und Testmuster der gerade getesteten Speicherzelle.
Bei Fehlern wird durch das Fehlerprotokoll die Art des Fehlers angezeigt.

3.6. Testdauer

Ein Durchlauf der Tests 0-8 dauert fuer 512 kB ca. 6 Minuten.

4. Kurzbeschreibung der Testmodule

4.1. Housenumber-Test

Der Housenumber-Test wird in mehreren Durchlaeufen ausgefuehrt. Im 1.Durchlauf - Lower Word Up - werden die Worte im angegebenen Speicherbereich mit dem unteren Wort ihrer 22 Bit-Adresse in aufsteigender Adressfolge beschrieben. Nach dem Schreiben eines Worts wird es geprueft; ist der ganze Speicherbereich beschrieben erfolgt eine zweite Pruefung mit der sich eine "Adressverkopplung abwaerts" innerhalb eines 64 KByte-Bereichs feststellen laesst.

Der 2.Durchlauf - Lower Word Down - fuehrt denselben Test mit abfallender Adressfolge durch, d.h. er beschreibt den Speicher von "oben" nach "unten". Auf diese Weise laesst sich eine "Adressverkopplung aufwaerts" erkennen.

Im 3. und 4.Durchlauf - Upper Word Up/Down - werden die Tests 1 und 2 mit dem oberen Wort der 22 Bit-Adresse durchgefuehrt, um Adressverkopplungen ueber die 64 KByte-Grenzen hinaus feststellen zu koennen.

Die Durchlaeufe 5-8 Lower Word Complement Up/Down und Upper Word Complement Up/Down - entsprechen den Tests 1-4, nur werden hier die komplentierten Daten verwendet, um auch Adressverkopplungen mit invertierten Bits erkennen.

4.2. Random-Pattern-Test

Im Random-Test wird der Speicherbereich mit einem reproduzierbaren "Zufalls"-Bitmuster wortweise beschrieben, wobei fuer jedes Wort ein neues Bitmuster berechnet wird. Danach erfolgt das wortweise Lesen und Vergleichen mit den erneut berechneten Bitmustern.

4.3. Bit-Shifting-Test

Der Bit-Shifting-Test besteht aus 34 Durchlaeufen in denen jeweils der ganze Speicherbereich mit demselben Bitmuster getestet wird. Der 1.Durchlauf verwendet das Bitmuster 1111111111111111111, der 2.Durchlauf das Bitmuster 11111111111111111110. Die Muster der naechsten 15 Durchlaeufe werden durch Linksschieben der 0 erzeugt. Anschliessend folgen weitere 17 Durchlaeufe mit den invertierten Bitmustern.

4.4. Refresh-Test

Beim Refresh-Test wird der Speicherbereich mit einem Schachbrett-Muster wortweise beschrieben und nach 5 Sekunden Wartezeit geprueft. Danach wird der Test mit dem invertierten Bitmuster wiederholt.

4.5. Opcode-Test

Beim Opcode-Test wird der Speicherbereich mit dem Opcode des RTS-Befehls (Return from Subroutine) beschrieben und anschliessend auf jedes Wort ein CALL-Befehl ausgefuehrt. Dann wird der Speicherbereich daraufhin geprueft, ob durch die Ausfuehrung der RTS-Befehle der Inhalt von Speicherzellen veraendert wurde.

4.6. Read/Modify/Write-Test

Der Read/Modify/Write-Test prueft den TAS-Befehl, der als einziger Befehl des MC68000 einen unteilbaren Zyklus fuer Lesen/Schreiben benutzt. Dazu wird der Speicherbereich byteweise geloescht und mit dem TAS-Befehl das Bit 7 auf 0 getestet und gesetzt (TAS = Test And Set). Mit einem zweiten TAS-Befehl wird geprueft, ob der erste TAS das Bit 7 gesetzt hat. Eine Fehlermeldung kann damit sowohl beim ersten TAS als auch beim Zweiten ausgegeben werden.

4.7. Write-Byte-Test

Der Write-Byte-Test beschreibt den Speicherbereich byteweise mit einem Schachbrettmuster und prueft jedes Byte sofort nach dem Schreiben.

4.8. Write-Long-Word

Der Write-Long-Word-Test beschreibt den Speicherbereich doppelwortweise mit einem Schachbrettmuster und prueft jedes Doppelwort sofort nach dem Schreiben.

4.9. Parity-Test

Der Parity-Test beschreibt im 1.Durchlauf den Speicherbereich wortweise mit einem Bitmuster mit gerader Bit-Anzahl. Danach wird jedes Wort gelesen und gleich wieder zurueckgeschrieben (Move memory to memory - Befehl). Das ist notwendig, da das Testprogramm im lokalen RAM ablaeuft und zur Erkennung eines Parity-Fehlers (Bus Error) zwei aufeinanderfolgende Q-Bus-Zyklen erforderlich sind, was bei dem verwendeten Befehl der Fall ist. Der 2.Durchlauf verwendet ein Bitmuster mit ungerader Bit-Anzahl.

4.10. Force-Parity-Test

Der Force-Parity-Error-Test setzt im Prozessor-Control-Register das FPE-Bit und beschreibt den Speicherbereich wortweise mit einem Bitmuster mit gerader Bit-Anzahl. Danach wird jedes Wort gelesen, wobei jeweils ein Parity-Fehler (Bus Error) auftreten muss, der vom Testprogramm abgefangen wird, sonst erfolgt eine Fehlermeldung. Dasselbe wird mit einem Bitmuster mit ungerader Bit-Anzahl wiederholt.

Programmbeschreibung

t68030

Prozessortestprogramm für den QU 68030 und QU 68050

Version 3.1

Martin Gröber
PCS GmbH

28. November 1983

1. Einleitung

Das Programm t68030 Version 3.1 ist aus dem Programm t68030 Version 1.1 entstanden. In der Version 3.1 wurden auch die Eigenschaften des Prozessors MC 68010 (Motorola) berücksichtigt. Der MC 68010 besitzt ein anderes Error-Handling als der MC 68000. Ausserdem wird beim Prozessor MC 68010 mit einer zusätzlichen Testroutine 'pf' die Page-Fault-Option überprüft. Für diesen Test ist der Prozessor QU68050 notwendig. Die zweite MMU-Stufe dieses Prozessors wird aber nicht ausgetestet.
Die Version 3.1 wird unter dem CADMUS Testmonitor aufgerufen und mit Parameter versorgt (siehe t68030(TM)).

2. Struktur des Programms

Das gesamte Programm besteht aus folgenden Moduln:

Hauptprogramm
10 Testroutinen
I/O-Routinen

3. Laden und Starten des Programms vom Testmonitor

@t68030 [param-list]

Für eine detailierte Beschreibung siehe t68030(TM).

4. Bedienung des Programms bei Aufruf mit dialog

Zunächst werden 2 Fragen an den Bediener gestellt:

Ob der Prozessor-Typ QU 68050 (mit MMU-Stufe 2) getestet werden soll.
In diesem Fall wird der MMU-Test (siehe Abschnitt 5) nicht durchlaufen.

Ob der verwendete Speicher die Erzwingung eines Parity-Fehlers (force parity error) ermöglicht.
Ist dies nicht der Fall, dann wird der Force-Parity-Error-Test (fp) ausgelassen.

Nun kann der Testlauf gestartet werden. Dabei gibt es im wesentlichen 2 unterschiedliche Betriebsarten:

single 1 Testdurchlauf

Die Testroutinen werden nur je 1x durchlaufen. Am Beginn und Ende einer jeden Routine und im Fehlerfall erscheint eine Meldung auf dem Bildschirm. Wahlweise ist hier auch eine Ausgabe auf Drucker möglich.

dauer Dauertest

Die Testroutinen werden fortlaufend nacheinander betrieben. Der Abbruch erfolgt durch Tastatureingabe oder wahlweise auch im Fehlerfall. Es wird im Normalfall nur die Anzahl der Testdurchläufe und die Anzahl der Fehler je Testroutine (wenn ungleich Null) angezeigt. Die Fehlerzähler werden pro Durchlauf um max. 1 erhöht (im Gegensatz zum Einzeltestdurchlauf, bei dem die genaue Anzahl der Fehler der jeweiligen Routine ausgegeben wird).

In beiden Betriebsarten kann im Fehlerfall der Test abgebrochen werden (Stop bei Fehler); es wird dann keine weitere Testroutine aufgerufen. Beim Dauertest kann dabei wahlweise die ausführliche Fehlerinformation angezeigt werden (wie beim Einzeltest).

Einige Testfunktionen können im Dauertest nicht durchgeführt werden (ROM-Test, BUS-INIT-Test). Deshalb sollten immer beide Betriebsarten benützt werden.

Alle Bediener-Eingaben können bis auf den 1.Buchstaben abgekürzt werden.

Durch Betätigen der Taste 'NO SCROLL' kann die Ausgabe auf das Terminal jederzeit unterbrochen werden (es wird auch die Ausführung des Programms angehalten). Der Smooth-Scroll-Mode des VT100 und DSG101 wird ebenfalls unterstützt.

5. Beenden des Programms

Funktion: 'ende' eingeben.
Die Kontrolle wird an den Testmonitor zurückgegeben.

6. Vorhandene Testroutinen

et Test der elementaren Prozessorfunktionen

Bedingte Verzweigungen, Compare-Befehle, Register-Test, byte-weise schreiben und lesen vom Speicher, TAS-Befehl mit dem Speicher (Read-Modify-Write-Zyklus)

ad Test der Adressierungsarten

Alle auf den Speicher wirkenden Adressierungsarten werden durch Schreib- und Lese-Zugriffe mit den Operandengrössen Byte, Wort und Doppelwort getestet.

be1 Test der 1-Operanden-Befehle

Alle 1-Operanden-Befehle (ausgenommen Sprung-, Verzweigungs-Befehle und Befehle mit Sonderfunktionen) werden mit verschiedenen Datenwerten und mit den Operandengrössen Byte, Wort und Doppelwort getestet. Die Operanden befinden sich im Datenregister 4 (ausser bei Shift-Befehlen). Nach der ausgeführten Operation wird auch der Inhalt des Condition-Code-Registers auf Richtigkeit überprüft. Im Fehlerfall werden der Befehl, der Operand, Soll- und Ist-Wert des Ergebnisses und des Condition-Code-Registers angezeigt.

be2 Test der 2-Operanden-Befehle

Alle 2-Operanden-Befehle (ausgenommen Immediate-Befehle, Befehle mit Sonderfunktionen und Befehle, die nur auf Adressregister wirken) werden mit verschiedenen Datenwerten und mit den Operandengrössen Byte, Wort und Doppelwort getestet. Die Operanden befinden sich in den Datenregistern 2 und 4. Nach der ausgeführten Operation wird auch der Inhalt des Condition-Code-Registers auf Richtigkeit überprüft. Im Fehlerfall werden der Befehl, die Operanden, Soll- und Ist-Wert des Ergebnisses und des Condition-Code-Registers angezeigt.

rr RAM/ROM-Test

Bei Einzeltestdurchlauf (single) wird über den ROM-Inhalt (alle 3 Teilbereiche) eine Prüfsumme erzeugt und ausgedruckt. Diese Prüfsumme darf sich nicht ändern, solange dieselbe Minitor-Version verwendet wird. Die Kontrolle muss manuell erfolgen.

Das interne RAM wird durch Einschreiben eines Testmusters (im 1. und 2. Testteil) bzw. durch Einschreiben der Adresse (im 3. Testteil) und auslesen des Inhalts überprüft.

Achtung, durch diese Veränderung des RAM-Inhalts wird die Information über evtl. gesetzte Breakpoints überschrieben.

mt MMU-Test

Beschreiben der Basisadressregister und lesen der erzeugten phys. Adresse über den Page-Deskriptor.

Es wird nur die MMU-Stufe 1 getestet. Beim Test des QU 68050 wird diese Routine nicht durchlaufen.

Beim QU 68030 kann das Page-Deskriptor-Register (PD) verwendet werden, um die durch die MMU erzeugte physikalische Adresse zu lesen.
Im Anlaufzustand, der hier beim Ansprechen des PD-Registers gewählt wird, sind für die Adressierung des PD-Registers die Bits 0...14 ausreichend. Die Bits 15...19 werden nur zur Bildung der phys. Adresse in der MMU herangezogen. Die Bits 20...23 wählen eines von 15 MMU-Segmenten aus (das Segment 0 wird hier nicht verwendet). Die Bits 9...21 der durch die MMU erzeugten phys. Adresse können aus dem PD-Register gelesen werden.

In diesem Programm werden die Basis-Adress-Register (BAR) und Addierer der MMU getestet durch Verwendung von 15 Segmenten, verschiedener BAR BAR-Inhalte und verschiedener Adressbit 15...19 des auf das PD-Register zugreifenden Befehls.

Segment 0 wird nicht getestet.

Beim Test des Segment 15 gelten folgende Einschränkungen:

Beim Segment 15 aufgetretene Fehler werden zwar registriert (der Fehlerzähler wird erhöht), beim Einzeldurchlauf erfolgt aber kein genauer Fehlerausdruck, da der Zugriff auf die DLV-11 über den Q-BUS nur mit ganz bestimmten BAR-Inhalten möglich ist.

Es können keine Breakpoints verwendet werden. Dies kann vermieden werden, indem der Test mit Segment 14 (statt 15) begonnen wird: Equate 'maxseg' entsprechend ändern.

sd Test der Segmentdeskriptoren der MMU

Die Segmentdeskriptoren werden beschrieben, Testzugriffe (Daten schreiben, Daten lesen, Code Fetch) auf das jeweilige Segment ausgeführt, die zum Teil gegen den gewählten Segment-Schutz verstossen (Trap!) und in diesen Fällen die Fehler-Anzeige im ESR kontrolliert.

Segment 0 wird nicht getestet.

Beim Test des Segment 15 gelten die gleichen Einschränkungen wie beim MMU-Test.

ir Interrupt-Test

Testet das Bus-Event-Signal (Linetime-Clock) und den Ausgabe-Interrupt vom Interface der Console.

In beiden Fällen wird der Interrupt zunächst durch eine höhere Priorität im Status-Register gesperrt, nach einiger Zeit die Priorität erniedrigt und die Zeit zwischen 2 Interrupts grob kontrolliert. (Die Baudrate des Terminals sollte deshalb zwischen 4800 und 19200 Bd liegen.)

Bei Einzeltestdurchlauf (single) wird zusätzlich das Bus-Init-Signal am Q-Bus getestet. Dies geschieht über das Interrupt-Enable-Bit des Consolen-Interface: Nach dem Bus-Init-Signal darf kein Interrupt mehr auftreten.

Im Fehlerfall wird eine Fehlertyp-Nummer angezeigt, sie hat folgende Bedeutung:

Fehlertyp

1	Interrupt von der Linetime-Clock ist aufgetreten, obwohl die Priorität im Statusregister höher sein sollte.
2...4	Der erwartete Interrupt von der Linetime-Clock ist ausgeblieben bzw. nicht innerhalb einer bestimmten Zeit gekommen.
5	Der Abstand zwischen 2 Interrupts von der Linetime-Clock war zu kurz.
6	Interrupt vom Consol-Interface ist aufgetreten, obwohl die Priorität im Statusregister höher sein sollte.
7...9	Der erwartete Interrupt vom Consol-Interface ist ausgeblieben bzw. nicht innerhalb einer bestimmten Zeit gekommen.
10	Der Abstand zwischen 2 Interrupts vom Consol-Interface war zu kurz.
11	Interrupt vom Consol-Interface kam, obwohl über den RESET-Befehl des 68000 ein Bus-Init ausgelöst wurde.
12	Interrupt vom Consol-Interface kam, obwohl über das Processor-Control-Register (PCR) ein Bus-Init ausgelöst wurde.

fp Force-Parity-Error-Test

Erzwingt durch 'force parity error' ein fehlerhaftes Parity-Bit im Speicher und kontrolliert, ob beim Auslesen dieser Zelle die richtige Fehlerreaktion erfolgt (Bus-Error, Inhalt des ESR).

Einige Speicher für den Q-Bus unterstützen diese Möglichkeit nicht; dies führt dann zu einer Fehleranzeige.

bt BUS-Timeout-Test

Erzwingt einen BUS-Timeout durch Beschreiben einer nicht vorhandenen Speicherzelle (Adresse E00000 hex) und kontrolliert die Fehlerreaktion. (Bus-Error, im ESR muss das Timeout-Bit oder das Page-Fault-Bit gesetzt sein.)

t68050

Kurzbeschreibung für MMU2-Testprogramm

Manfred Glöckel
PCS GmbH

4. März 1983

1. Voraussetzung

1.1. Hardware

CADMUS System
QU 68050 Prozessor
mindestens 512 kByte Speicher
Serielle Schnittstelle
FLoppyDisk oder anderer Massenspeicher
Terminator

1.2. Software

CADMUS Testmonitor
Testprogramm t68050

2. MMU2-Test: t68050

2.1. Laden und Starten des Programms vom Testmonitor

@*t68050* [param-list]

Für eine detailierte Beschreibung siehe t68050(TM).

2.2. Laden und Starten des Programms durch den Minitor

.rx (laden von der Floppy)
./t68050
.g (Start)
(Der Punkt ist das Minitor-Prompt-Symbol.)

2.3. Programmausgabe:

Memory von xxx bis xxx (hex)
loop xxx error xxx

Erklärung zum Ausdruck:

Loop gibt die Anzahl der Durchläufe an.
error gibt die Summe der Fehler an.
Memory von x bis x verfügbarer Testspeicher

3. Kurzbeschreibung des Programms

Die Grösse des Testspeichers wird angezeigt. Innerhalb des Testspeichers werden virtuell 16Mbyte dargestellt.

Das Programm testet ob beim Beschreiben des Seitendiskriptors die beiden Statusbits gleich 0 und bei einem entsprechenden Bus-Zyklus (Lese-, Schreibzugriff) auf 1 gesetzt wurden. Wird auf eine gerade nicht im Speicher

befindliche Seite oder auf eine nicht definierte Seite zugegriffen wird ein Page-fault ausgeloest.

Kachel 0 - 12 und 1023 werden nicht getestet.

Bei loop 0,1,2 ...,usw wird die Adresse (loop * 4) innerhalb jeder Kachel getestet.

4. Fehlermeldung

z.B.

*** bus error nicht eingetroffen
kachel = xxx viradr = xxx pd = xx -> aaa -> bbb
loop xxx error xx

kachel	Kachelnummer
viradr	virtuelle Adresse
pd	Inhalt des Page-diskriptors
-> aaa	Zugriff (Test, Lese-, Schreibzugriff)
-> bbb	Testsollstatus der Kachel
loop	Anzahl der Durchläufe
error	Summe der Fehler

kaadr	Kacheladresse
mmu3	Inhalt der mmu3 (offset)

5. Programmabbruch

Das Programm kann durch CTRL-C abgebrochen werden.

tdma

Kurzbeschreibung für DMA-Testprogramm des QU68030/50

Martin Gröber

PCS GmbH

1. März 1983

1. DMA-Test: DMATST

1.1. Laden und Starten des Programms vom Testmonitor

@tdma [param-list]

Für eine detailierte Beschreibung siehe tdma(TM).

1.2. Laden und Starten des Programms durch den Minitor

.rx (laden von der Floppy)
./tdma
.g (Start)
(Der Punkt ist das Minitor-Prompt-Symbol.)

1.3. Bedienung des Programms bei Aufruf mit dialog

Es gibt 3 unterschiedliche Betriebsarten. Diese sind: Read, Write und Move. Möglich sind auch Kombinationen der drei verscheidenen Betriebsarten.

Testmemory von XXX bis YYY (hex) (j/n)

Die Grösse des Speichers wird überprüft. Bei Eingabe von <n> kann ein anderer zu testender Speicherbereich ausgewählt werden. Eingaben sind dann: Startadresse und Endadresse des zu testenden Speichers. Die Grösse des zu testenden Speichers muss mindestens 2000 (hex) Byte sein. Nach der Eingabe wird überprüft, ob der Speicher wirklich im System ist und zur Bestätigung nochmals abgefragt. Bei Fehlern in der Eingabe wird eine neue Eingabe erwartet.

DMA-Read (j/n) Im DMA-Betrieb wird blockweise vom Speicher gelesen.

DMA-Write (j/n) Im DMA-Betrieb wird blockweise in den Speicher geschrieben.

DMA-Move (j/n) Im DMA-Betrieb wird vom Speicher gelesen und auf den Speicher geschrieben.

Dauertest (j/n) Bei ja befindet sich das Programm in einer Endlosschleife die nur durch *CTRL C* unterbrochen werden kann. Bei nein wird nur ein Durchlauf gemacht.

DER Eingabe des zu verwendenden DMA-Extensionregisters. Befindet sich am Backplane keine Zusatzverdrahtung muss das DER 0 verwendet werden.

DMA-Req.Time Hier kann die Anzahl der DMA-Requests pro Sec. eingegeben werden. Mögliche Eingaben sind 5 - 255 ms.

Programmausgabe:

Loop xx Ges.Error xx DMA-Time xxx

		akt	buserr	checks	intimo	daterr	adress	wcnt
read	*	*	xx	xx	xx	xx	xx	xx
writ	*	*	xx	xx	xx	xx	xx	xx
move	*	*	xx	xx	xx	xx	xx	xx

Erklaerung zum Ausdruck:

Loop	Anzahl der Durchläufe
akt	gerade aktive Funktion
buserr	Anzahl der aufgetretenen Busfehler
checks	Anzahl der aufgetretenen Checksummen-Fehler
intimo	Anzahl der Timeout-Fehler
daterr	Anzahl der Datenfehler
adress	Adresse bei der ein Datenfehler aufgetreten ist
wcnt	Wordcount bei aufgetretenem Datenfehler

1.4.

Das Programm kann durch *CTRL C* abgebrochen werden.

Beschreibung
für
SCSI-Testprogramm

Periphere Computer Systeme
Pfälzer-Wald-Str. 36
8000 München 90

1. Einleitung

Das Programm testet die Schreib-, und Lesefunktion des Omti 20d bzw. DTC 520A Controllers in Verbindung mit 5 1/4" Winchester-, oder Floppylaufwerken. Ein Controller kann max. zwei Winchester und zwei Floppys bedienen. Es werden sektorweise Daten auf die Floppy bzw. Winchester geschrieben und wieder gelesen. Hierzu wird nicht der UNIX-Treiber, sondern eigene Subroutines benutzt. Durch Datenvergleich und durch Abfrage der Controller Fehleranzeigen oder durch Timeout werden Schreib-, Lese-, und Datenfehler erkannt.

2. Voraussetzung

2.1. Hardware

CADMUS System
Serielle Schnittstelle
FLoppy Disk oder anderer Massenspeicher
OMTI 20D oder DTC 520 Controller und 5 1/4 " Winchester
SCSI - Adapter
Adresse: FFFB80(16) Vektor: 230(8)
Terminator

2.2. Software

CADMUS Testmonitor
Testprogramm SCSITST

"SCSITST" ist ein in C geschriebens Testprogramm, das mit dem Testmonitor gestartet wird.

3. Bedienerführung

3.1. Laden und Starten des Testprogrammes

@scsitst [param-list]

Für eine detailierte Beschreibung siehe SCSITST(TM).
Bei Start mit dialog meldet sich das Programm mit folgender Systemausgabe.

*** OMTI 20D DTC-520A Test-Programm ***

0 = OMTI 20D
1 = DTC-520A
enter controller type

Abfrage nach welchem SCSI-Controller

0 = TM 503
1 = TM 603
2 = TM 703
3 = floppy
4 = RO 208
5 = MAX 1065
6 = BASF 6185
enter drive type

Abfrage nach welchem Massenspeicher

unit
0 1 = Winchester
2 .. 3 = Floppy
enter unit [0..3]

Abfrage nach welchem Winchester-, bzw. Floppy-Laufwerk. 0=1., 1=2. Winchester-Laufwerk, 2=1., 3=2. Floppy-Laufwerk.

Das Programm bietet mehrere Testmodule zur Auswahl an.

Testprogramm-Auswahlliste:

0 = read block
1 = write block
2 = read sequent
3 = write sequent
4 = write + read + compare sequent
5 = write + read + compare zigzag
6 = write sequent read + write random
enter test number

Hier können die verschieden Tests ausgewählt werden.

test with move multiple [y/n]

Uebertragungsmodus wort-, bzw. blockweise.

continous test [y/n]

Es besteht die Möglichkeit, das Programm im Dauertest zu betreiben.

3.2. Programm-Abbruch

Das Programm kann jederzeit mit "CTRL C" abgebrochen werden.

3.3. Fehler- und Statusreport (S)

Während des Programmlaufes wird immer angezeigt, welcher Teiltest gerade läuft. Durch Drücken der S-Taste wird eine Statuszeile ausgegeben. Die Ausgabe enthält: Test-Nr., Anzahl der Durchläufe und Summe aller Fehler.

3.4. Anhalten der Bildschirmausgabe (W)

Um ein Weglaufen des Bildschirminhaltes zu verhindern wird mit der W-Taste die Ausgabe angehalten. Nach Erkennen der Wait-Funktion wird dies mit "continue <cr>" am Bildschirm angezeigt. Mit der Return-Taste wird die Ausgabe wieder fortgesetzt.

4. Kurzbeschreibung der Testmodule

4.1. Read Block - Write Block

Bei Read Block bzw. Write Block kann ein beliebiger Sektor innerhalb der angegebenen Grenzen ausgewählt werden. Bei ungültiger Sektorgrösse wird die Abfrage wiederholt. Nach dem Lesen bzw. Schreiben wird getestet ob ein Fehler aufgetreten ist.

4.2. Read sequent Write sequent

Auf der Winchester bzw. Floppy werden fortlaufend alle Sektoren mit einem festen Datenmuster beschrieben bzw. gelesen.

4.3. Write Read Compare sequent

Auf der Winchester bzw. Floppy wird fortlaufend ein Zylinder beschrieben, gelesen und die gelesenen Daten mit der Sollinformation verglichen. Zwischen Schreiben und Lesen findet keine Kopfbewegung statt.

4.4. Write Read Compare zigzag

Vom diesem Test werden die Sektoren alternierend (1. Sektor, letzter Sektor, 2. Sektor, usw) ausgewählt, beschrieben, gelesen und mit der Sollinformation verglichen.

4.5. Write sequent Read Write random

Das zu testente Medium wird sequentiell beschreiben. Danach wird nach Zufall ein beliebiger Sektor ausgewählt, gelesen und mit der Sollinformation verglichen. Nun wird der Sektor mit eimem anderen Muster beschrieben.

Beschreibung
für
SLS 88 - Testprogramm

Periphere Computer Systeme
Pfälzerwald Str. 36
8000 München 90

1. Einleitung

Das Programm "SLUTST" testet das SLS-48 bzw. SLS-88 Interface im Kurzschluss. Der Test arbeitet mit Interrupts. Es wird mit Baud-Raten zwischen 110 Baud und 9600 Baud getestet.

2. Voraussetzung

2.1. Hardware

CADMUS System (CPU + Speicher)
Serielle Schnittstelle DLV11
FLoppy Disk oder anderer Massenspeicher
SLS 48 oder SLS 88
Adresse: FFFC00(16) Vektor: 300(8)
Terminator

2.2. Software

CADMUS Testmonitor
Testprogramm SLUTST

"SLUTST" ist ein in C geschriebenes Testprogramm, das mit dem Testmonitor gestartet wird.

3. Bedienerführung

3.1. Laden und Starten des Testprogrammes

@slutst [param-list]

Für eine detailierte Beschreibung siehe SLUTST(TM).
Bei Start mit dialog meldet sich das Programm mit folgender Systemausgabe.

S L U T E S T P R O G R A M M
Keyboard Commands:
^C : Abbruch S : Show W : Wait <Ret> : Continue

Schaltereinstellung: S1 S8 = off
S9 ... S0 = on

Schaltereinstellung ok ?

Nach Ueberprüfung des 10 pol. Miniaturschalters auf korrekte Einstellung ist diese Abfrage mit Return zu beantworten.

Konfigurationsregister: 9600 Baud, 8bit, 1Stop bit sls 88

Hier wird der Inhalt des Konfigurationsregisters angezeigt und getestet ob es sich um ein sls-48 oder sls-88 Interface handelt.

Dauertest [J/N]

Bei Dauertest gleich 'J' für ja werden alle Tests im Endlos-loop durchgeführt.

Auto-Kurzschlusstest [J/N]

Bei Auto-Kurzschlusstest müssen folgende Verbindungen hergestellt werden.

Transmitter: tty40 - Receiver: tty42
Transmitter: tty42 - Receiver: tty40
Transmitter: tty41 - Receiver: tty43
Transmitter: tty43 - Receiver: tty41
Transmitter: tty44 - Receiver: tty46
Transmitter: tty46 - Receiver: tty44
Transmitter: tty45 - Receiver: tty47
Transmitter: tty47 - Receiver: tty45

Verbindungen ok ?

Nach Ueberprüfung der vorgeschriebenen Verbindungen ist dies mit Return zu quittieren.

Bei Nicht-Auto-Kurzschlusstest können beliebige Verbindungen hergestellt werden.

Sendekanal [tty4x]
Empfangskanal [tty4x]

X gibt den Kanal (0 .. 7) an. Bei Eingabe von Ctrl C Return wird zur Sendekanalabfrage verzeigt.

3.2. Programm-Abbruch (^C)

Das Programm kann jederzeit mit "Contr C" abgebrochen werden.

3.3. Fehler- und Statusreport (S)

Während des Programmlaufes wird immer angezeigt, welcher Teiltest gerade läuft. Durch Drücken der S Taste wird eine Statuszeile ausgegeben. Die Ausgabe enthält: Anzahl der Testdurchläufe, Summe aller Fehler, Interruptfehler, Vectorfehler und Datenvergleichsfehler.

3.4. Anhalten der Bildschirmausgabe (W)

Um ein Weglaufen der Terminalinformation zu verhindern wird mit der W-Taste die Ausgabe angehalten. Nach erkennen der Wait-Funktion wird dies mit "continue <Ret>" am Bildschirm angezeigt. Mit der Return-Taste wird die Ausgabe wieder fortgesetzt.

4. Kurzbeschreibung der Testmodule

4.1. Auto-Kurzschlusstest

Bei Auto-Kurzschlusstest werden die oben angegebenen Kanäle der Reihe nach mit den Baud-Raten 9600 Baud, 4800 Baud, 2400 Baud, 1200 Baud, 600 Baud, 300 Baud und 110 Baud getestet. Als Datenmuster werden alle möglichen Zeichen von 00(hex) bis FF(hex) übertragen. Am Terminal wird die zu testende Verbindung und die augenblickliche Geschwindigkeit angezeigt.

4.2. Nicht-Auto-Kurzschlusstest

Bei Nicht-Auto-Kurzschlusstest kann eine beliebige Verbindung zwischen einem Transmitter und einem Receiver hergestellt werden.

5. Systemmeldungen

5.1. loop error cmperr vecerr interr

loop	Anzahl der Durchläufe
error	Summe aller Fehler
cmperr	Anzahl der Datenvergleichsfehler
vecerr	Anzahl der Vectorfehler
interr	Anzahl der Interruptfehler

5.2. Data Error soll = aa ist = ba cmperr = x

Datenvergleichsfehler mit Sollmuster gleich *aa* und Empfangmuster gleich *ba*. *cmperr* = x ist der x. Vergleichsfehler in diesem Block.

5.3. Receiver Interrupt Timeout /dev/tty4z xxxx Baud

Bei Kanal z(0..7) ist kein Zeichen empfangen worden.

5.4. Transmitter Interrupt Timeout /dev/tty4z xxxx Baud

Nach Starten des Kanals z(0..7) ist kein Sendeinterrupt eingetroffen.

5.5. Interrupts: Transmit = x Receive = y Ext/Status = z Special = w

x	Anzahl der Tramsmitter Interrupts
y	Anzahl der Receiver Interrupts
w	Anzahl der Ext/Status Interrupts
w	Anzahl der Special Interrupts

5.6. Special Receive Condition status bits:

10(hex)	Parity Error
20(hex)	Rx Overrun Error
40(hex)	Framming Error

5.7. External status bits:

40(hex)	Tx Underrun
80(hex)	Break/Abort

FPP 81

FLOATING POINT PROZESSOR

Testanweisung

28. Oktober 1983

PCS GmbH

INHALTSVERZEICHNIS

1. Einleitung

Diese Testanweisung setzt die Kenntnis folgender Beschreibungen voraus:

Spezifikation FPP 81 Floating Point Prozessor
Hardware-Beschreibung FPP 81
Datenblatt NS16081-6 (-10) Floating Point Unit
CADMUS Testmonitor Benutzeranleitung

2. Voraussetzungen

2.1 Testhilfsmittel

CADMUS-System mit Q22-Bus
Minitor Version 2.2 (R900.123)
FPP 81 + S-Bus-Kabel
DLV 11 oder Multi-Function-Board
Platten-, Band- oder Streamer-Laufwerk
Platten-, Band- oder Streamer-Controller
Terminal

2.2 Testprogramm

fpptst

3. Testablauf

3.1 Sichtkontrolle

- Ist das Board vollständig bestückt?
- Befinden sich alle IC's am richtigen Steckplatz?
- Stimmt die IC-Richtung?
- Sind alle Elkos richtig gepolt ?
- Sind alle Brücken richtig konfiguriert?

3.2 Kurzschlussprüfung

Messen der 5 Volt Versorgungsspannung auf Kurzschluss

3.3 Basistests mit Minitor

Um die Durchführung einfacher Tests ohne Logic-Analyser zu ermöglichen, wurden auf dem FPP 81 ein Test-Register, sowie verschiedene Test-Sequenzen in den Sequenzer-PROM's vorgesehen.

3.3.1 Test-Register

Das 16 Bit breite Testregister speichert jedes Kommandowort, das an die FPU übergeben wird, bis zum nächsten Befehl. Damit kann durch Auslesen des Testregisters geprüft werden, ob das FPU-Kommandowort aus der FPP-Adresse richtig umcodiert wurde. Der Zusammenhang zwischen FPP-Adresse und Kommandowort ist der Spezifikation (Bild 3.3 ff.) zu entnehmen.

Das Testregister kann per Minitor unter der Adresse XXFFF8 (hex) ausgelesen werden. XX ist die Basisadresse des FPP, die Standardeinstellung ist 3E (hex).

Beispiel:

3E0020.2 <CR>	FPP-Befehl ROUNDFB
3EFFF8.2 <CR>	Testregister lesen
2402	Ausgabe

3.3.2 Test-Sequenzen

Die Testsequenzen im Sequenzer ermöglichen einen Datentransfer vom Operanden-Buffer in den Result-Buffer, ohne Beteiligung der FPU. Die Umschaltung in den Testbetrieb des Sequenzers erfolgt über das TST-Bit im Command/Status-Register, das mit folgender Minitor-Eingabe zu setzen ist:

3EFFF0-20

Führt man jetzt per Minitor einen FPP-Schreibzugriff durch, so erfolgt der Transfer von 1,2 oder 4 Worten vom Operanden- in den Result-Buffer. Die Anzahl der Worte hängt vom jeweiligen Befehl ab. Wählt man z.b. einen MOVFF-Befehl (mit ext. Source), so erfolgt ein 2-Wort-Transfer, bei einem MOVLL-Befehl erfolgt ein 4-Wort-Transfer.

Beispiele:

Schreibzugriff mit 1-Wort-Transfer

3E2800-1234 <CR>	FPP-Befehl MOVWF
3EFFE0.2 <CR>	Result-Buffer lesen
1234	Ausgabe

Schreibzugriff mit 2-Wort-Transfer

3E9000-1111 2222 <CR>	FPP-Befehl MOVFF
3EFFE0.4 <CR>	Result-Buffer lesen
1111 2222	Ausgabe

Schreibzugriff mit 4-Wort-Transfer

3ED000-1111 2222 3333 4444 <CR>	FPP-Befehl MOVLL
3EFFE0.8 <CR>	Result-Buffer lesen
1111 2222 3333 4444	Ausgabe

3.4 Testprogramm 'fpptst'

3.4.1 Uebersicht

Der Floating Point Prozessor FPP 81 ist als peripherer Prozessor am QU68000 angeschlossen und wird mit MOVE-Befehlen entweder über den Q-Bus oder über den S-Bus angesprochen.

Das Testprogramm 'fpptst' dient als GO/NOGO Test und läuft im Standalone-Betrieb oder unter dem CADMUS Testmonitor. Es können damit Einzel- und Dauertests durchgeführt werden. Der FPP sollte zuerst ohne S-Bus-Anschluss getestet werden. Erst wenn damit alle Testdurchläufe fehlerfrei sind, ist der S-Bus anzuschliessen.

Interrupt-Vektor: 231 (oktal) FPP-Basisadresse: 3E0000

3.4.2 Beschreibung

Folgende Tests werden bei einem Testdurchlauf ausgeführt :

1. Befehls-Tests (Tests 1..19,21,22)

   Alle Floating-Point-Befehle (ausser Double-Precision) werden mit verschiedenen Adress-Modes überprüft.

2. Compare(CMP)/CSR-Test (Test 20)

   In diesem Test werden der COMPARE-Befehl und die Statusbits der FPU (Negative und Zero), die bei diesem Befehl ins CSR übertragen werden, geprüft.

3. Adress-Mode-Tests

   - *Schreibzugriff:*

   Source extern, Destination intern (Test 23)

   Source intern, Destination extern (Test 24)

   - *Lesezugriff:*

   Source intern, Destination extern (Test 25)

   Source intern, Destination intern (Test 26)

   Source intern, Destination intern (Test 27)

   - *Registerpointertests:* (Tests 23..27)

   Die Schreib/Lese-Zugriffe (auf Reg. F0) werden wiederholt, wobei der interne Operand jeweils durch den Registerpointer adressiert wird. Für den Registerpointer werden die Modes (RP), -(RP), --(RP), (RP)+ und (RP)++ verwendet. Beim Test 27 erfolgt eine Kombination aller Register-Pointer-Modes untereinander (z.b. -(RP),(RP)++).

4. TRAP-Test (Test 28)

   In diesem Test wird die Trap-Condition der FPU überprüft. Dër Trap wird durch einen ZERO DIVIDE ausgelöst, wobei ein Interrupt erzeugt werden muss.

5. DOUBLE-PRECISION-Test (Test 29)

   In diesem Test wird der Befehl MOVLL (mit Double-Precision-Zahl) geprüft.

6. OP.BUFFER-RES.BUFFER-TRANSFER-Test (Test 30)

   In diesem Test wird mit Hilfe einer Testsequenz im Sequenzer die Datenübertragung vom Op-Buffer in den Res-Buffer (ohne Beteiligung der FPU) geprüft.

Die vorhandenen Tests kann man entweder im Single- oder im Loop-Betrieb starten. Im Loop-Betrieb kann der Test durch Eingabe von Ctrl Z gestoppt werden, durch Eingabe des Kommandos 'e' wird das Testprogramm abgebrochen.

Die Fehler, die während der Tests auftreten, werden durch Fehlerzähler und Fehlermeldungen auf dem Bildschirm oder Drucker protokolliert. Der Abbruch des Testprogramms im Fehlerfall kann wahlweise angegeben werden.

3.4.3 Bedienung

Mögliche Kommandos:

Sxy	Einmalige Durchführung	des Test Nr. xy
Szz	Einmalige Durchführung	aller Tests
Lxy	Dauertest	des Test Nr. xy
Lzz	Dauertest	aller Tests
Cxy	Lösche Test Nr. xy	
G	Start	
E	Exit	

zz = (maximale Anzahl der Tests) + 1

xy = maximal zweistellige Zahl zwischen 0 und (zz-1), z.B <01>,< 2>,<3>

3.4.4 Fehlermeldungen

Tests 10,11,14,15,16,17:

Zusätzlich zum Fehlerzähler wird eine Fehlermeldung im folgenden Format ausgegeben

"FPU :aaaa aaaa I1:I2 bbbb bbbb -- cccc cccc QU68 :dddd dddd"
(FPU-Ergebnis,Input-Operand1,Input-Operand2,QU68000-Ergebnis)

Das QU68000-Ergebnis wird per Software berechnet und mit dem FPU-Ergebnis verglichen.

Tests 23..27:

Zusätzlich zum Fehlerzähler wird ein Fehlercode (z.b. 420) mit folgender Bedeutung ausgegeben :

Fehlerbit	Adress-Mode	Fehler
0 (1)	Fx	Daten
1 (2)	Fx	Reg.Ptr.
2 (4)	RP	Daten
3 (8)	RP	Reg.Ptr.
4 (10)	RP+	Daten
5 (20)	RP+	Reg.Ptr.
6 (40)	RP++	Daten
7 (80)	RP++	Reg.Ptr.
8 (100)	-RP	Daten
9 (200)	-RP	Reg.Ptr.
10 (400)	--RP	Daten
11 (800)	--RP	Reg.Ptr.

Fx = FPU-Register x
RP = Register-Pointer

Bei Fehler-Abbruch:

Der gesamte Zustand des FPP wird am Bildschirm ausgegeben, d.h. es erfolgt ein Dump von Op-Buffer, Res-Buffer, aller FPU-Register, CSR und Test-Register.

3.4.5 Beispiele

Laden des Testprogramms mit dem Minitor von Floppy:

.rx
./fpptst
.g0

Laden des Testprogramms mit dem Testmonitor:
fpptst dialog

Dialog:
Gib Kommando [max. 3 Zeichen]:L1
Gib Kommando [max. 3 Zeichen]:L 2
Gib Kommando [max. 3 Zeichen]:L3
Gib Kommando [max. 3 Zeichen]:L15
Gib Kommando [max. 3 Zeichen]:g
Konsole [J/N]: J
LP-Drucker [J/N] : N

Die Tests 1,2,3, und 15 werden als Dauertest gestartet.
Abbruch mit ^Z

Gib Kommando [max. 3 Zeichen]:L31
Gib Kommando [max. 3 Zeichen]:c21
Gib Kommando [max. 3 Zeichen]:c22
Gib Kommando [max. 3 Zeichen]:g
Konsole [J/N]: J
LP-Drucker [J/N] N

Alle Tests mit Ausnahme von Test 21 und 22 werden als Dauertest gestartet.
Abbruch mit ^Z

Gib Kommando [max. 3 Zeichen]:s10
Gib Kommando [max. 3 Zeichen]:s 9
Gib Kommando [max. 3 Zeichen]:s20
Gib Kommando [max. 3 Zeichen]:s07
Gib Kommando [max. 3 Zeichen]:g
Konsole [J/N]: j

LP-Drucker [J/N] : n

Der Test 7 wird einmal gestartet.

Weiter [J/N] ? : j (Test 9 ?)
Weiter [J/N] ? : j (Test 10?)
Weiter [J/N] ? : j (Test 20?)
Weiter [J/N] ? : j (Test 7 ?)
Weiter [J/N] ? : j (Test 9 ?)
Weiter [J/N] ? : n (Test 10?)
Gib Kommando [max. 3 Zeichen]:e
PROGRAMM-ENDE

LBP68000

LASER PRINTER SYSTEM

Testanweisung

Bernhard Rothballer 1.März 1983

1. Einleitung

Für den Test des LBP68000-Gesamtsystems wurde ein Standalone-Testprogramm entwickelt, das verschiedene Testmuster erzeugt und auf dem Laser-Drucker ausgibt. Damit kann die Funktion des LBP-KE und des Laser-Druckers sowie die Druckqualität überprüft werden, was immer nach Reparaturen, Wartungs- und Einstellarbeiten am Drucker erforderlich ist.

2. Voraussetzungen

2.1 Dokumentation

Diese Testanweisung setzt die Kenntnis folgender Beschreibung voraus:

LBP-KE Hardware/Firmware-Dokumentation

Drucker-Service-Unterlagen:

CANON LBP10 Operation Manual
CANON LBP10 Service Handbook
CADMUS Testmonitor Benutzeranleitung

2.2 Testhilfsmittel

QU68000-System mit Q22-Bus
LBP-KE Laser Printer Controller bestehend aus
LBP-CP B907.034 mit PROM-Satz gebr. nach R900.062
LBP-MA B922.202 (LBP10-MIKIBUS-Adapter)
Verb.kabel K900.452
Verb.kabel K924.302
512 KByte RAM
DLV 11 oder Multi-Function-Board
Platten-, Band- oder Streamer-Laufwerk
Platten-, Band- oder Streamer-Controller
Terminal (VT100 komp.)
Laser Printer LBP10

2.3 Testprogramm

lbptst

3. Testablauf

3.1 Testprogramm *lbptst*

Das Testprogramm *lbptst* wird vom Minitor oder vom Testmonitor geladen und gestartet. Es meldet sich mit einem Menue der verfügbaren Testkommandos bzw. -Muster.

3.1.1 Testmuster

Gibt man die Nummer eines Testmusters ein, kommt nach einigen Sekunden, die der Rechner zum Aufbau des Bildes im Speicher braucht, die Abfrage nach der Anzahl der Kopien; nach dieser Eingabe sollte der Drucker anlaufen (falls nicht ☛ 3.1.4). Eine Kombination aller Testmuster wird mit dem Kommando "K" ausgedruckt.

3.1.2 Statusabfrage

Mit dem Kommando "S" kann der Status des Laser-Druckers abgefragt werden, der als Hex-Code am Bildschirm ausgegeben wird (Normal: 001BH). Zur Auswertung des Codes siehe Hardware/Firmware-Dokumentation.

3.1.3 Fehlermeldungen

Bei Auftreten bestimmter Fehler, z.b Drucker nicht selektiert, erfolgt eine Fehlermeldung mit Angabe des Command/Status-Register-Inhalts. Zur Auswertung des Codes siehe Hardware/Firmware-Dokumentation.

3.1.4 Fehler-Diagnose

Bleibt das Testprogramm hängen und/oder läuft der Drucker nicht an, kann mit Hilfe des Controller-Diagnose-Registers der Zustand des LBP-KE festgestellt werden. Die Zustandscodierung ist der Hardware/Firmware-Dokumentation zu entnehmen. Wartet z.b das LBP-KE vergeblich auf ein bestimmtes Signal vom Drucker, kann aus dem Zustandscode entommen werden, um welches Signal es sich handelt. In jedem Fall sollte bei Auftreten derartiger Fehler ein INIT und Neustart des Testprogramms versucht werden.

Beschreibung
für
MUX-KE - Testprogramm

Periphere Computer Systeme
Pfälzer-Wald-Str. 36
8000 München 90

1. Einleitung

Mit Hilfe des Testprogramms *muxketst* kann das MUX-KE-Modul der Firma PCS getestet werden. *Muxketst* ist speziell auf den Prozessor QU68000 (Q68030/50) zugeschnitten.

2. Voraussetzung

2.1. Hardware

- MUX-KE Modul
  Adresse: 766000

- 1 bis 4 DLV11-j
  Adressen: 776500
  776540
  776600
  776640

- Kurzchluss-Stecker für V24:

output	input
=====	=====
s---------	r
r---------	s
masse---------	masse

2.2. Software

- CADMUS Testmonitor
- Testprogramm MUXKETST

Muxketst ist ein Standalone-Testprogramm das ohne Betriebssystem abläuft. Gestartet wird *muxketst* mit dem Testmonitor. Optional kann *muxketst* auch direkt vom Minitor geladen und gestartet werden (siehe Punkt 3).

3. Testprogramm muxketst

3.1. Laden und Starten des Programms vom Testmonitor

@muxketst [param-list]

Für eine detailierte Beschreibung siehe MUXKETST(TM).

3.2. Laden und Starten des Programms durch den Minitor

```
.40.7ffff=000000<cr>   (Optional)
.rx                    (Laden von der Floppy)
./muxketst
.g0                    (Start)
```

(Der Punkt ist das Minitor-Prompt-Symbol.)

3.3. Programmausgabe:

```
***    MUX-KE TEST     V01.xx     ***
1  Senden     2 : Kurschluss
L3,S3: ALL  G: START   Cxy: CLEAR   E: EXIT
-----------------------------------------
Gib Kommando [ maximal 3 Zeichen ]:
```

Die vorhandenen Tests kann man entweder als Einzel- oder Dauertest starten. Als mögliche Kommandos sind einzugeben:

Sxy	Einmalige Durchführung des Tests *xy*
S3	Einmalige Durchführung aller Tests
Lxy	Test *xy* im Endlos-Betrieb
L3	Endlos-Betrieb aller Tests
Cxy	Lösche Test *xy*
G	Start
E	Rückkehr zum Testmonitor

xy ist maximal eine zweistellige Zahl zwischen 1 und 2, z.B.:

L01<cr>	Dauerbetrieb für Test 1
L02<cr>	Dauerbetrieb für Test 2
C	Test 2 nicht durchgeführt werden
S2<cr>	Einmalige Durchführung von Test 2

Beispiel: Alle Tests im Endlos-Betrieb starten

```
L3<cr>
G<cr>
```

3.4. Programmeingabe

3.4.1. Minitor-Betrieb

Bei Aufruf von *muxketst* direkt vom Minitor erfolgt die Testauswahl wie oben beschrieben. Besondere Parameter werden im Dialog wie beim Aufruf des Programmes vom Testmonitor mit dem Parameter **dialog** eingegeben.

3.4.2. Testmonitor-Betrieb

3.4.2.1. Testauswahl durch Aufruf:

muxketst<cr>	Default Test 1 und 2
muxketst test=1<cr>	nur Test 1
muxketst test=2<cr>	nur Test 2
muxketst test=1,2<cr>	Test 1 und 2

Für weitere Möglichkeiten siehe MUXKETST(TM).

3.4.2.2. Dialog

Bei Aufruf mit dem Parameter **dialog** führt das Programm folgenden Dialog mit dem Bediener:

1) Konsole [J/N]?:
   J: Das Protokoll wird auf der Konsole erstellt.
   N: Kein Protokoll auf der Konsole. Diese Option ist bei HW-Messungen zu empfehlen, da die Testfolge schneller durchgeführt wird.
2) LP-Drucker [J/N]?:
   J: Das Protokoll kann zusätzlich zur Terminal-Ausgabe auf einem Drucker ausgegeben werden. Bei Dauertest werden nur die Fehlermeldungen ausgegeben.
3) Gib Anzahl der Bytes:
   Hier kann man eine Zahl zwischen 0 und 255 eintippen. Diese Zahl gibt an, wieviele Zeichen gesendet werden sollen.
4) Gleiches Datum {x} [J/N]?:
   J: Nur x's werden gesendet. Das x ist ein beliebiges ASCII- Zeichen.
   N: Die Zeichen werden in aufsteigenden Form gesendet, z.B: abcdef.
5) Gib OUTPUT channel [0..15]:
   Hier muss man die Nummer des Sendekanals eingeben.

Kanal 0 bis 3:	1. DLV11-J Modul mit der Adresse: 776500
Kanal 4 bis 7:	2. DLV11-J Modul mit der Adresse: 776540
Kanal 8 bis 11:	3. DLV11-J Modul mit der Adresse: 776600
Kanal 12 bis 15:	4. DLV11-J Modul mit der Adresse: 776640

6) Gib INPUT channel [0..15]:
   Hier muss man die Nummer des Empfängerkanals eingeben. Siehe 5)

3.5. Statusreport

Tritt während des Tests ein Fehler auf, wird dieser Fehler durch einen Zähler auf dem Bildschirm oder auf dem Drucker protokolliert.

LOOP-COUNTER entspricht der Anzahl der Tests, die durchgeführt wurden.

ERROR-COUNTER gibt die Anzahl der fehlerhaften Versuche bei den entsprechenden Tests an. Z.B.:
Anzeige: 1: 100 2: 300
Interpretation:
Test 1 hat bisher 100 fehlerhafte Versuche. Test 2 hat bisher 300 fehlerhafte Versuche.

3.6. Fehlermeldungen

Fehler, die vom MUX-KE-Modul gemeldet werden (Siehe MUX-KE Multiterminal-Controller Beschreibung D920.221):

Transmit error
Parity error
Framing error
Data overrun
Silo overflow
Non existent memory

Diese Meldungen zeigen die Anzahl der Fehler an, die während des Tests auftreten, z.B.:
Anzeige: *Parity error: 40*
Interpretation: Es haben bisher 40 Uebertragungen mit Parity-Fehler stattgefunden.

Fehler, die der Test selbst feststellen kann:

Receive-Timeout	Hier empfängt man die Daten entweder zu langsam oder überhaupt nicht.
Send-Timeout	Der Sender ist entweder zu langsam, oder beendet sich nicht innerhalb eines festgelegten Zeitraums.
Too many received bytes	Hier wurden mehr Zeichen empfangen als erwartet.
Too less received bytes	Hier wurden weniger Zeichen empfangen als erwartet.

Diese Meldungen zeigen die Anzahl der Fehler an, die während des Tests auftreten, z.B.:
Anzeige: *Too less received bytes: 10*
Interpretation: Hier haben bisher 10 Uebertragungen mit zu wenig empfangenen Daten stattgefunden.

Vergleich error gibt an, wieviele Zeichen bei einem vorangegangenen Versuch fehlerhaft empfangen wurden (nur bei Test 2 möglich).

3.7. Programmabbruch

Durch Eingabe von *CTRL-C* kann das Programm jederzeit abgebrochen werden.

4. Kurzbeschreibung der Testmodule

Sende-TEST:

- Für Minitor-Betrieb: (Kommandos: s1, l1, s3, l3)
- Hier werden auf einen Kanal (OUTPUT CHANNEL) nur Daten gesendet. Alle möglichen Sendefehler werden auf der Konsole protokolliert.

Kurschluss-TEST:

- Für Minitor-Betrieb: (Kommandos: s2, l2, s3, l3)
- Hier werden zuerst Daten über einen Kanal gesendet und über einen anderen empfangen. Dafür wird ein entsprechender Kurzschluss-Stecker zwischen beiden Kanälen benötigt.

Beschreibung
für
DZV11 - Testprogramm

Periphere Computer Systeme
Pfälzer-Wald-Str. 36
8000 München 90

1. Einleitung

Mit Hilfe des Testprogramms ***dztst*** kann das DZV11-Modul der Firma PCS getestet werden. *Dztst* ist speziell auf den Prozessor QU68000 (Q68030/50) zugeschnitten.

2. Voraussetzung

2.1. Hardware

- DZV11 Modul (4 oder 8 Kanäle)

  Basisadresse: 0xffe040 Vektor: 0360

- Kurzchluss-Stecker für V24: (DEC: H329)

```
output        input
======        =====
      s---------r
      r---------s
    DTR---------CO,RI
   CO,RI--------DTR
  masse---------masse
```

2.2. Software

- CADMUS Testmonitor
- Testprogramm DZTST

Dztst ist ein Standalone-Testprogramm das ohne Betriebssystem abläuft. Gestartet wird ***dztst*** mit dem Testmonitor. Optional kann ***dztst*** auch direkt vom Minitor geladen und gestartet werden (siehe Punkt 3).

3. Testprogramm dztst

3.1. Laden und Starten des Programms vom Testmonitor

@dztst [param-list]

Für eine detailierte Beschreibung siehe DZTST(TM).

3.2. Laden und Starten des Programms durch den Minitor

```
.40.7ffff=000000<cr>   (Optional)
.rx                    (Laden von der Floppy)
./dztst
.g0                    (Start)
```

(Der Punkt ist das Minitor-Prompt-Symbol.)

3.3. Programmausgabe:

```
***    DZV11 TEST     V01.xx     ***
1 Senden     2 : Kurschluss  3: Modem (DTR=>CO,RI)
L4,S4: ALL  G: START   Cxy: CLEAR   E: EXIT
-----------------------------------------
Gib Kommando [ maximal 3 Zeichen ]:
```

Die vorhandenen Tests kann man entweder als Einzel- oder Dauertest starten. Als mögliche Kommandos sind einzugeben:

Sxy	Einmalige Durchführung des Tests *xy*
S4	Einmalige Durchführung aller Tests
Lxy	Test *xy* im Endlos-Betrieb
L4	Endlos-Betrieb aller Tests
Cxy	Lösche Test *xy*
G	Start
E	Rückkehr zum Testmonitor

xy ist maximal eine zweistellige Zahl zwischen 1 und 3, z.B.:

L01<cr>	Dauerbetrieb für Test 1
L02<cr>	Dauerbetrieb für Test 2
C	Test 2 nicht durchgeführt werden
S2<cr>	Einmalige Durchführung von Test 2

Beispiel: Alle Tests im Endlos-Betrieb starten

```
L4<cr>
G<cr>
```

3.4. Programmeingabe

3.4.1. Minitor-Betrieb

Bei Aufruf von *dztst* direkt vom Minitor erfolgt die Testauswahl wie oben beschrieben. Besondere Parameter werden im Dialog wie beim Aufruf des Programmes vom Testmonitor mit dem Parameter **dialog** eingegeben.

3.4.2. Testmonitor-Betrieb

3.4.2.1. Testauswahl durch Aufruf:

dztst<cr>	Default Test 1, 2 und 3:
dztst test=1<cr>	nur Test 1
dztst test=2<cr>	nur Test 2
dztst test=1,2<cr>	Test 1 und 2

Für weitere Möglichkeiten siehe DZTST(TM).

3.4.2.2. Dialog

Bei Aufruf mit dem Parameter **dialog** führt das Programm folgenden Dialog mit dem Bediener:

1) Konsole [J/N]?:
   J: Das Protokoll wird auf der Konsole erstellt.
   N: Kein Protokoll auf der Konsole. Diese Option ist bei HW-Messungen zu empfehlen, da die Testfolge schneller durchgeführt wird.

2) LP-Drucker [J/N]?:
   J: Das Protokoll kann zusätzlich zur Terminal-Ausgabe auf einem Drucker ausgegeben werden. Bei Dauertest werden nur die Fehlermeldungen ausgegeben.
3) Gib Anzahl der Bytes:
   Hier kann man eine Zahl zwischen 0 und 255 eintippen. Diese Zahl gibt an, wieviele Zeichen gesendet werden sollen.
4) Gleiches Datum {x} [J/N]?:
   J: Nur x's werden gesendet. Das x ist ein beliebiges ASCII- Zeichen.
   N: Die Zeichen werden in aufsteigenden Form gesendet, z.B: abcdef.

3.5. Statusreport

Tritt während des Tests ein Fehler auf, wird dieser Fehler durch einen Zähler auf dem Bildschirm oder auf dem Drucker protokolliert.

LOOP-COUNTER entspricht der Anzahl der Tests, die durchgeführt wurden.

ERROR-COUNTER gibt die Anzahl der fehlerhaften Versuche bei den entsprechenden Tests an. Z.B.:
Anzeige: 1: 100 2: 300
Interpretation:
Test 1 hat bisher 100 fehlerhafte Versuche. Test 2 hat bisher 300 fehlerhafte Versuche.

3.6. Fehlermeldungen

Fehler, die vom DZV11-Modul gemeldet werden (Siehe DZV11, Digital microcomputer interfaces handbook):

Transmit error
Parity error
Framing error
Data overrun
Silo overflow
Non existent memory

Diese Meldungen zeigen die Anzahl der Fehler an, die während des Tests auftreten, z.B.:
Anzeige: *Parity error: 40*
Interpretation: Es haben bisher 40 Uebertragungen mit Parity-Fehler stattgefunden.

Fehler, die der Test selbst feststellen kann:

Receive-Timeout	Hier empfängt man die Daten entweder zu langsam oder überhaupt nicht.
Send-Timeout	Der Sender ist entweder zu langsam, oder beendet sich nicht innerhalb eines festgelegten Zeitraums.
Too many received bytes	Hier wurden mehr Zeichen empfangen als erwartet.
Too less received bytes	Hier wurden weniger Zeichen empfangen als erwartet.

Diese Meldungen zeigen die Anzahl der Fehler an, die während des Tests auftreten, z.B.:
Anzeige: *Too less received bytes: 10*

Interpretation: Hier haben bisher 10 Uebertragungen mit zu wenig empfangenen Daten stattgefunden.

Vergleich error gibt an, wieviele Zeichen bei einem vorangegangenen Versuch fehlerhaft empfangen wurden (nur bei Test 2 möglich).

3.7. Programmabbruch

Durch Eingabe von *CTRL-C* kann das Programm jederzeit abgebrochen werden.

4. Kurzbeschreibung der Testmodule

Sende-TEST:

- Für Minitor-Betrieb: (Kommandos: s1, l1, s4, l4)
- Hier werden auf einen Kanal (OUTPUT CHANNEL) nur Daten gesendet. Alle möglichen Sendefehler werden auf der Konsole protokolliert.

Kurschluss-TEST:

- Für Minitor-Betrieb: (Kommandos: s2, l2, s4, l4)
- Hier werden zuerst Daten über einen Kanal gesendet und über einen anderen empfangen. Dafür wird ein entsprechender Kurzschluss-Stecker zwischen beiden Kanälen benötigt.

Hinweise zur Fehlerursache, -behebung und Teststrategie am System C A D M U S

Wenn die Fehlerursache bei einem Absturz mit Register-Dump nicht selbst behoben werden kann, senden Sie bitte eine ausgefuellte Kopie des beigefuegten Dump-Formblatts an den PCS Service, Muenchen.
Zur Analyse eines Register-Dumps beachten Sie bitte auch CRASH(8) im MUNIX Manual I (ab V1.5)

Fehlerzustaends des Rechners	moegliche Ursachen	erkennbar an folgen-typischen Merkmalen	Abhilfe
keinerlei Reaktion auf der Konsole nach dem Einschalten	Sicherungen durchgebrannt	Luefter steht	Sicherung austauschen
	Prozessor defekt		Karte austauschen
	Terminal -Leitung defekt		Leitungen ueberpruefen
	-Terminal defekt		Ersatz-Terminal
	-falscher SET-UP-mode		Einstellung ueberpruefen
	serielle Konsol-Schnittstelle defekt		Karte austauschen
	Bus-Verlaengerung falsch gesteckt (bei Erweiterungsbox)		Anschluss der Bus-Verlaengerung kontrollieren
keine Meldung .MINITOR	Speicher -fehlt	Cursor springt um 2 Positionen nach rechts	Speicherkarte austauschen
	-defekt		
	-Adressen falsch eingestellt		Adressen korrigieren
Schmutz-Zeichen auf der Konsole	Setup-mode des Terminals falsch		Setup-Mode neu einstellen
	serielle Konsol-Schnittstelle defekt		Karte austauschen
Keine Terminal-Eingabe moeglich trotz Meldung .MINITOR	Terminal-Rueckleitung defekt		Leitung ueberpruefen
	Transmit-Baudrate falsch eingestellt		Setup-mode ueberpruefen
	Keyboard defekt		Keyboard austauschen
	serielle Konsol-Schnittstelle (Eingang) defekt		Karte austauschen
kein laden von /unix, /nodename/unix oder Standalone-programmen moeglich	Platte noch nicht hochgefahren	Meldung: hkerror bn=2 cs=2 err=....	10 sec warten, dann erneuter Versuch
	Stoerungen an der Platte:		
	-Kabelverbindung nicht ok	Emulex-controller:	ueberpruefen

		Kontroll-Lampe blinkt	
	-Platten-Transport- Sicherung nicht geloest	Ready-Lampe am Plattenlaufwerk (soweit vorhanden oder zugaenglich) leuchtet nicht	Sicherung loesen
	Filesystem zerstoert	Meldung: can't find file	Platte formatieren; dann "SET-UP-MUNIX"
	Platte nicht formatiert		
	Bad-Block in der file /unix	Meldung: hkerror bn=dddd...	/sa/boot laden, dann hk(0,0)/unix laden
	falsches Boot-Device verwendet	Meldung: can't find file	Versuch, von alternativem Datentraeger zu booten
		bus error esr=1	
	Minitor passt nicht fuer das Boot-Device		s.o. oder Minitor-PROM austauschen
keine Reaktion nach 'g0'	Speicher nicht in Ordnung Winchester oder Controller defekt		Testmonitor laden: Speichertest mit "memtst" Plattentest mit "check" Prozessortest mit "t68030" oder Austausch von Speicher, Plattencontroller, Prozessor
keine weitere Terminal-Ausgabe nach "etart of c 68000 Clear Memory ... free memory = ... largest contiguous block = ... "	Swap-bereich kleiner als Hauptspeicher		mit aunix hochfahren, dann unix neukonfigurieren
	UNIX falsch konfiguriert		
	eines der folgenden Programme fehlt: /etc/init /etc/gtty /bin/sh		Root-Filesystem restaurieren bzw. 'SET-UP-MUNIX'
	kein Interrupt vom Platten-Controller Speicher oder Prozessor defekt		Testmonitor laden: Speicher-, Prozessor- und Plattentest
Absturz mit Register-dump	daisy-chain unterbrochen	exception 24 in Register-Dump	HW umstecken (daisy-chain schliessen)
(siehe dazu auch CRASH(8) in Manual I ab MUNIX V1.5)	Speicher-Timeout	bus-error ... esr=1 in Register-Dump	Speicher und Prozessor testen
	MUNIX falsch konfiguriert	bus-error ... esr=1 access address=ff....	mit /aunix hochfahren; Konfiguration mit Kommando "whatconf" ueberpruefen evtl. unix neu konfigurieren
	ein Controller hat sich	dito	Ueberpruefen, ob Controller

	gelockert oder ist defekt		richtig stecken; Controller testen
	Swap-Bereich zu klein	Meldung: panic: out of swap-space	Unix mit groesserem Swap-Bereich konfigurieren
	MUNIX falsch konfiguriert oder Root-Fs zerstoert	Meldung: panic: no fs oder panic: iinit	mit /aunix hochfahren Unix neu konfigurieren oder Root-Fs restaurieren
	Speicher oder Prozessor defekt	illegal opcode	Speicher- und Prozessor testen
	Prozessorhalt (evtl ausgeloest durch Break-taste an der Systemkonsole oder durch unbeabsichtigtes Druecken der Run-Lampe)	exception 30	Breaktaste an der Systemkonsole disablen Schluesselschalter auf LOCK stellen
	Interrupt-Vektor auf einem Controller falsch eingestellt	illegal vector interrupt	Interrupt-Vektor auf Controller korrigieren oder /usr/sys/l.s anpassen und neues unix generieren
double-bus-error, d.h. Busfehler waehrend einer Busfehler-Behandlung	Unix-Konfiguration passt nicht zu einer vorhandenen DMA-Verdrahtung ein 22-bit Device-Treiber wird fuer einen 18-bit Controller verwendet oder umgekehrt	kein Echo an der Systemkonsole bei Eingabe (run-Lampe erloschen)	mit aunix hochfahren; Unix entsprechend der DMA-Verdrahtung neu konfigurieren mit aunix hochfahren; lib3 in /usr/sys anpassen (aus libchoice die passenden Treiber nach lib3 bringen) Unix neu konfigurieren
feck laeuft nicht	Stackgize von /etc/feck zu klein /etc/feck fehlt oder zerstoert File /etc/fstab falsch (/etc/checklist in V1.5)	unter /etc wird das File core erzeugt Meldung: cannot open /dev/...	Stackgize erhoehen mit /bin/stkgiz feck von einem back-up-Medium restaurieren fstab (checklist in V1.5) korrigieren
/etc/feck liefert Inkonsistenzen	File-System teilweise zerstoert	siehe feck-Beschreibung	Filesystem reparieren mit feck oder komplett restaurieren
Absturz beim Uebergang in den multi-user-mode	Zugriff auf nicht vorhandene Controller (fuer Terminals und/oder Drucker)	im Register-dump: buserror ... esr=1 access address=ff....	Controller nachruesten oder im Single User Mode hochfahren und die Dateien /etc/ttys und /etc/rc ueberpruefen. In /etc/ttys die angeschlossenen Terminals korrekt eintragen. (/etc/ttys ist in V1.5 ersetzt durch /etc/inittab)
	MUNIX falsch konfiguriert	dito	im Single User Mode mit whatconf die Konfiguration ueberpruefen (auf dz dh und lp achten) Unix neu konfigurieren

Sehr lange Reaktions zeiten nach Uebergang in Multi User Mode	die special-files (unter /dev) fuer die tty's sind falsch generiert	Prozessnummern wachsen rasend an	mit kill -1 1 in den Single User Mode schalten (bzw. mit init s ab V1.5), dann im Directory /dev: make dh (bei MUX-KE) make dz (bei DZV11) File /etc/ttys (/etc/inittab ab MUNIX V1.5) ueberpruefen und eventuell korrigieren
nur Konsole arbeitet im Multi User Mode	die special-files (unter /dev) fuer die tty's fehlen oder sind falsch falsche Eintraege im File /etc/ttys (/etc/inittab ab MUNIX V1.5)		special files wie oben korrigieren File /etc/ttys (/etc/inittab ab MUNIX V1.5) ueberpruefen und korrigieren
doppeltes Echo bei eingegebenen Zeichen auf manchen Terminals	UNIX liest sowohl vom MUX-KE Zeichen ein, als auch von DLV11J	doppeltes Echo (ausser an der Systemkonsole) (nur wenn MUX-KE vorhanden)	File /etc/ttys falsch: Wenn fuer tty10-tty23 ein login-Prozess kreiert wird (z.B. 1vtty12) darf fuer tty1-tty7 kein login-Prozess kreiert werden (z.B. 0vtty3) /etc/ttys korrigieren (/etc/ttys ist in MUNIX V1.5 durch /etc/inittab ersetzt)
RK disk error RL " RM "	bad block vorhanden	Meldung: no replace sector available	betroffenes Plattendrive mit "check" testen; evtl. bad sector manuell eintragen, anschliessend unter Unix File-System-Check
System steht nach Uebergang von Multi in Single User Mode	ein Prozess kann nicht abgebrochen werden: - ein Interrupt von einem peripheren Geraet steht aus - fuer die MUX-KE-Leitung, die von der Systemkonsole verdeckt wird, wurde ein login-Prozess kreiert	keine Reaktion an der Systemkonsole nach kill -1 1 (bzw. init s ab V1.5)	Init-Taste betaetigen Unix hochfahren und fsck ausfuehren vor Uebergang in den Multi User Mode das File /etc/ttys ueberpruefen kein login-Prozess fuer tty13 oder tty23, je nachdem ob die Konsole auf der 1. oder 2. DLV11J liegt (/etc/ttys ist in MUNIX V1.5 durch /etc/inittab ersetzt)

MUNIX-Systemabsturz am ..... um ....., aufgezeichnet von

vermutete Absturzursache bzw. Verdacht: .....
.....

Absturzprotokoll:

registers	A0	A1	A2	A3	
	A4	A5	A6	A7	US .....
	D0	D1	D2	D3	
	D4	D5	D6	D7	

mmu[1-14]

*Text-Fehlerursache:

cpustate ..... *access address ..... instruction register

status reg ..... *pc ..... *ssr

*procedure addresses : lineno (bilden Sie bitte die Prozedur-Adressen mit Hilfe von /usr/sys/unix.sym auf Prozedurenamen ab)

.....!.....	.....!.....	.....!.....	.....!.....
.....!.....	.....!.....	.....!.....	.....!.....
.....!.....	.....!.....	.....!.....	.....!.....
.....!.....	.....!.....	.....!.....	.....!.....

mina7

Hinweise: Die Werte der Register, die nicht auf der Systemkonsole angezeigt werden, sind gleich 0.

Es ist nicht notwendig alles von der Systemkonsole abzuschreiben. Die wichtigsten Informationen, die unbedingt notiert werden sollen, sind mit * markiert.

Bad Sector Handling

Version 1.2

PREFACE

Magnetic disks are derived into seperate sectors by the controller. Due to error conditions some sectors might be unsuitable for read and write operations (bad sectors). If the number of bad sectors is below a pre-defined limit, the disk can still be used under the operating system MUNIX. In this case, MUNIX ensures that bad sectors are replaced by error-free sectors.

During normal execution, bad sector handling is transparent to the user. It supports new magnetic disks, containing bad sectors on delivery and bad sectors which came about as a result of continuous use.

The following magnetic disk types are supported: Standard-RL01/02, Standard-RK06/07 and Standard-RM02/03/05.

Author's initials: RW

Trademarks:

MUNIX, CADMUS	for PCS
DEC, PDP	for DEC
UNIX	for Bell Laboratories

TABLE OF CONTENTS

1. INTRODUCTION

The bad sector handling routine is divided into the following three areas:

- Disk Organization,
- Disk Driver,
- Verification Programs.

1.1 Disk Organization

Each disk is regarded as a linear sucession of "n" sectors. The sectors are numbered "0" through to "n-1". A disk is divided into the following three areas:

- common area,
- replacement sectors area,
- bad sector information file.

Each area may contain bad sectors. The common area is used by MUNIX for the definition of file systems. Bad sectors found in this area cause sectors to be used in the replacement area. The bad sector file contains a directory of all bad sectors found on the disk and a reference to the replacement sectors. For each bad sector found in the replacement area, an additional sector is used within this area. The bad sector file is based on a redundant structure, i.e. information stored in bad sectors in this area is not destroyed.

1.2 Disk Driver

There are two versions of the disk driver: the stand-alone version and MUNIX-version. For bad sector handling, both versions must be able to perform the following functions:

1. Protection against unauthorized access on that part of the disk reserved for bad sector handling (replacement area, bad sector file).

2. User transparent replacement of bad sectors with replacement sectors.

1.3 Verification Programs

Special utility programs have been developed to check magnetic disks for bad sectors and to initialize the bad sector file for the individual magnetic disk.

On delivery, new magnetic disks should be checked for bad sectors. Additional bad sectors can be generated on disks already used. In this case, the sector must be marked appropriately and the information transferred to a replacement sector.

1.4 User Information

When using magnetic disks, the user (system administrator) should to adhere to the following approach:

1. A new disk should be verified for bad sectors and the bad sector file should be initialized (Verification Program).

2. Now the magnetic disks can be used under the operating system MUNIX.

If a bad sector is found during operations, the following steps need to be taken:

3. The information stored in the bad sector is to be transferred to a replacement sector (Verification Program). The information contained in the bad sector is destroyed.

4. The user can continue to use the magnetic disk under the operating system MUNIX.

1.5 Compatibility to Earlier MUNIX Releases

RL01/02 and RK06/07 magnetic disks, used in earlier MUNIX releases not containing bad sector handling routines (inclusive V1.4) are not compatible with subsequent releases. The magnetic disks involved can be used in future, if the following steps are executed:

1. safe magnetic disk contents with the operating system release used so far,

2. check magnetic disk for bad sectors,

3. the contents are now to be written back, using the new release.

Please note that the file sizes might be changed (see rl(4), hk(4)).

Disk types RM02/03 are compatible if addressed via the hp-driver (RP04/05/06, RM02/03 without bad sector handling). However, they can be converted in the way described, so that the rm-driver fitted with bad sector handling can be used. In this case, the contents of the magnetic disk are to be saved with the hp-driver, after which the contents are to be restored using the rm-driver. File sizes do not change for RM02/03 magnetic disks (see hp(4), rm(4)).

2. SPECIFICATION

2.1 Disk Organization

2.1.1 Division of Areas

The magetic disk division in common area, replacement area and bad sector file is stated in detail in the following table.

AREA	First Sector	Last Sector	No. of Sectors
Total disk	0	n-1	n
Common area	0	p-1	p
Replacement area	p	r-1	r-p
Bad sector file	r	n-1	nr

The disk organization and the bad sector file structure is very similar to the DEC Standard 144. The size of each area is specified in a way that the maximum of 126 bad sectors can be handled. Reference is also made to the rough division in tracks and cylinders.

The bad sector file is, for example, always allocated to the last track of the last cylinder. Due to the fact that individual disk type vary in track size (number of sectors per track), the sizes of the replacement areas and bad sector files are dependent on the disk type used. The structure of the magnetic disks currently supported is found in the following table.

	NUMBER OF SECTORS			
Disk Type	Total Disk	Common Area	Replacement Area	Bad Sector File
RL01	20480	20280	160	40
RL02	40960	40760	160	40
RK06	27126	26972	132	22
RK07	53790	53636	132	22
RM02/03	131680	131520	128	32
RM05	500384	500224	128	32

2.1.2 Bad Sector File

The bad sector file is structured as follows:

Sector	Contents
0	bad sector information
1	
2	not used
3	
4	copy from 0, 1
5	
6	not used
7	
8	copy from 0, 1
9	
10	not used
11	
12	copy from 0, 1
13	
14	not used
15	
16	copy from 0, 1
17	
18	not used
19	

The bad sector information consists of 256 16-bit words:

Word	Contents	
0	0	
1	0	
2	Number of bad sectors	
3	0	
4 5	Sector number	1st bad sector entry
6 7	Sector number	2nd bad sector entry
252 253	Sector number	125th bad sector entry
254 255	Sector number	126th bad sector entry

The bad sector table containing sector numbers is named "bad sector table". Unused bad sector elements are filled with "-1". On each magnetic disk there are five copies of the bad sector table available. A disk can be used at any time if one of the copies does not contain bad sector entries and no more than 126 bad sectors are found on the disk.

2.1.3 Allocation: Bad Sector - Replacement Sector

The allocation bad sector - replacement sector" is determined by the sequence of entries found in the bad sector table:

> The last sector available in the replacement area (sector number r-1) is allocated to the first bad sector found (1st bad sector entry); the last replacement sector but 1 (r-2) is allocated to the second bad sector found, etc.

2.1.4 Allocation: Sector Number - MUNIX Blocknumber

MUNIX data blocks are 1024 bytes long, whereas disk sectors are 256 bytes (RL01/02) or 512 bytes (RK06/07, RM02/03/05) long. Therefore, a MUNIX block spans 2 to 4 sectors.

The stand-alone driver considers a disk as a linear succession of blocks, numbered 0 to n/4-1 or n/2-1 (n = number of disk sectors). The allocation is (dependent on the sector size):

Sector Size	Block Number	Sector Number(s)
256 bytes	a	4*a - 4*a+3
512 bytes	a	2*a - 2*a+1

When using MUNIX, a disk can be subdivided into a number of file systems. The file system blocks are numbered sequentially, relative to the first block of the file system (starting with 0). The absolute block number "a" of a particular block within a given file system is calculated by adding all blocks allocated to prior file systems (offset) as well as the relative block number within this file system (r). By taking the absolute block number "a", the user receives the allocated sector number (see table above).

2.2 Verification Programs

The verification programs are responsible for the following tasks:

- find all bad sectors on a disk,
- generate the bad sector table and initialize the bad sector file on disk.

A test run needs to be performed on disk, in order to find the bad sectors.

2.2.1 Sector Test

The following types of sector tests are available:

1. Write/Read processing cycle with subsequent comparison operations. If an error is indicated by the disk controller, the write/read operation is not repeated. A sector is regarded as "bad", if a write/read error is detected or the comparison result is found to be negative. If possible, the header of the individual sector is marked accordingly.

2. Optionally, a sector test is possible using the "read-only-mode". In this case the disk content is not overwritten. After a read error is detected, the relevant sector is declared "bad".

The bad sector table is generated at will. New bad sector entries are added to the end of the table. Since the sequence of entries also indicates the allocation of replacement sectors, it is possible in this way to enter bad sector table entries subsequently. The sequence of "old" bad sectors is not changed.

2.2.2 Test Runs

Verification programs provide a number of functions:

1. Complete test (bad sector scan):
   The complete disk is tested. In the first step, all sectors are overwritten in ascending order with the DEC test pattern "0xed6db6db". In the second step, all sectors are read in reverse order. The read and write operations are performed on complete tracks. If an error is found, each sector is tested individually and the bad sector is determined. In this operation, the bad sector file is generated and written to disk, when the test is terminated.

Functions 2 to 4 assume that the bad sector file is already available on disk.

2. Partial test:
   A contiguous number of sectors (specifications are entered in dialogue mode) are tested. Prior to the actual test, the bad sector file is read. Alternating continuously (first sector, last sector, second sector, last but one sector, ...) the sectors are tested with a random pattern. During the test operation some new entries might be put into the bad sector table, in which case the table is written back to the bad sector file prior to terminating the test program.

3. Manual entries of bad sectors:
   If a disk is used frequently and bad sector develop during normal usage, in some cases the verification program is not able to detect all bad sectors. This function helps to identify such cases. The bad sector file is read and the user enters the numbers of the bad sectors involved. The sectors are marked "bad", and the sector identifications are entered in the bad sector table. Subsequently the bad sector table is written back onto disk.

4. Listing all known bad sectors:
   The bad sector file is read and the identification numbers of all bad sectors known are listed.

A further function is available for extented tests:

5. Random sector test:
   A random character generator produces a sequence of sector numbers. The sectors addressed by these numbers are tested, using the random pattern generated by the test program.

   The test runs on a continuing basis but can be terminated with a "bus reset".

   Attention: If a bad sector file is stored on disk, the file may be destroyed.

2.3 Disk Driver

The general approach of both stand-alone and MUNIX driver are similar:

> Normal write/read access operations are performed as long as no bad sector is detected. (Bad sector detection is dependent on the disk type used; further details are found below.)
>
> If the bad sector table has not yet been read (initialization status or disk change), it is now read into memory. If the sector is marked "bad", the system accesses the relevant replacement sector. If the following message is displayed on the system console: **disk error.**

Bad sector detection:

a) The sector is marked "bad" in the header (e.g. RK06/07, RM02/03/05):
In a write/read operation, the disk controller produces an error message. Each time the System tries to access a bad sector the controller answers with an error message, causing the driver to access the relevant replacement sector.

b) The sector cannot be marked in the header (e.g. RL01/02):
In order to avoid writing on a sector while not being able to read it without error, access to bad sectors are not allowed on principle. Prior to a write/read operation, the bad sector table is checked to find out if the required sector is marked "bad". In this case, the system immediately accesses the replacement sector.

   This approach assumes that the bad sector table is read into memory prior to the first disk access or immediately after a disk is exchanged.

The stand-alone driver is different to MUNIX drivers, in that the first disk access reads the bad sector table. For this reason, the stand-alone program is to be restarted immediately after a disk is exchanged. Operating under MUNIX however, disks can be exchanged at any time without re-booting the system.

Bad Block Behandlung

Version 1.2

Abstract

Magnetplatten werden vom Controller in einzelne Sektoren unterteilt. Ein Teil dieser Sektoren kann unbrauchbar sein (*bad sectors*). Liegt die Anzahl der bad sectors unter einer bestimmten Obergrenze, kann die Platte trotzdem unter MUNIX verwendet werden. MUNIX sorgt in diesem Fall dafür, dass bad sectors durch Ersatzsektoren ersetzt werden.

Dieses *bad sector handling* ist während des normalen Betriebs für den Benutzer transparent. Es unterstützt sowohl fabrikneue Platten mit bad sectors als auch Platten bei denen während des Betriebs neue bad sectors entstehen.

Zur Zeit werden folgende Plattentypen unterstützt:
Standard-RL01/02, Standard-RK06/07 und Standard-RM02/03/05.

Autoren-Kennzeichen: RW

Bad Sector Handling V1.2

Rudolf Wildgruber
PCS Gmbh
Pfälzer-Wald-Str. 36
8000 München 90

1. Einführung

Das *bad sector handling* teilt sich auf in drei Teilbereiche:

- Plattenorganisation
- Disk-Check-Programme
- Plattentreiber

1.1. Plattenorganisation

Jede Platte wird als lineare Folge von *n* Sektoren betrachtet. Die Sektoren sind von *0* bis *n-1* durchnumeriert. Eine Platte wird in drei Bereiche eingeteilt:

- öffentlicher Bereich
- Bereich für Ersatzsektoren
- bad sector Information *(bad sector file)*

Jeder dieser Bereiche darf bad sectors enthalten. Der öffentliche Bereich wird unter MUNIX für das Anlegen von Filesystemen verwendet. Für bad sectors in diesem Bereich werden im Ersatzbereich Sektoren belegt *(Auslagerung)*. Das bad sector file enthält ein Verzeichnis aller bad sectors der Platte und die Verweise zu den Ersatzsektoren. Für einen bad sector im Ersatzbereich wird ein weiterer Sektor in diesem Bereich belegt. Das bad sector file ist redundant aufgebaut, so dass bad sectors in diesem Bereich in der Regel die benötigte Information nicht vernichten.

1.2. Plattentreiber

Ein Plattentreiber existiert in zwei Versionen: *Standalone-Version* und *MUNIX-Version*. Beide Versionen müssen für das bad sector handling folgende Funktionen gewährleisten:

a) Schutz des für das bad sector handling reservierten Teils der Platte (Ersatzbereich, bad sector file) vor unberechtigtem Zugriff.

b) Für den Benutzer transparente Ersetzung von bad sectors durch ihre Ersatzsektoren.

1.3. Disk-Check-Programme

Spezielle Dienstprogramme übernehmen die Aufgabe, eine Platte auf bad sectors zu überprüfen und das bad sector file dieser Platte zu initialisieren.

Bei fabrikneuen Platten muss die gesamte Platte auf bad sectors überprüft werden. Bei bereits beschriebenen Platten bei denen ein weiterer bad sector festgestellt wird, muss dieser entsprechend markiert und ausgelagert werden, ohne dass die restliche Information auf der Platte zerstört wird.

August 23, 1983

1.4. Benutzersicht

Der Benutzer (Systemverwalter) hat beim Einsatz einer Platte folgendermassen vorzugehen:

1) Überprüfung einer neuen Platte auf bad sectors und Initialisierung des bad sector files (Disk-Check-Programm)
2) Benutzung der Platte unter MUNIX

und falls ein weiterer bad sector während des Betriebs auftritt:

3) Auslagerung des bad sectors in den Ersatzbereich (Disk-Check-Programm). Die Information, die der bad sector enthielt, ist zerstört.
4) Weitere Benutzung der Platte unter MUNIX.

1.5. Kompatibilität zu früheren MUNIX-Releases

RL01/02- bzw. RK06/07-Platten, die unter einer früheren MUNIX-Release ohne bad sector handling (bis einschl. V1.4) in Gebrauch waren, sind nicht kompatibel zu den nachfolgenden Releases. Man kann diese Platten weiterhin verwenden, wenn der Inhalt der Platte mit der bisherigen Release gerettet, die Platte auf bad sectors überprüft und der Inhalt mit der neuen Release wiederhergestellt wird. Es ist zu beachten, dass sich die Filesystem-Grössen eventuell ändern (siehe *rl(4)*, *hk(4)*).

RM02/03-Platten sind weiterhin kompatibel, wenn sie über den hp-Treiber (RP04/05/06, RM02/03 ohne bad sector handling) angesprochen werden. Sie können jedoch in der geschilderten Art und Weise auf den rm-Treiber mit bad sector handling umgestellt werden. Dazu ist der Inhalt einer Platte mit dem hp-Treiber zu retten und mit dem rm-Treiber wiederherzustellen. Die Filesystem-Grössen ändern sich bei RM02/03-Platten nicht (siehe *hp(4)*, *rm(4)*).

2. Spezifikation

2.1. Plattenorganisation

2.1.1. Einteilung in Bereiche

Die Einteilung einer Platte in öffentlicher Bereich, Ersatzbereich und bad sector file ergibt sich aus nachstehender Tabelle.

Bereich	erster Sektor	letzter Sektor	Sektorzahl
gesamte Platte	0	n-1	n
öffentlicher Bereich	0	p-1	p
Ersatzbereich	p	r-1	r-p
bad sector file	r	n-1	n-r

Die Plattenorganisation und der Aufbau des bad sector file lehnt sich eng an den DEC Standard 144 an. Die Grösse der einzelnen Bereiche wird für die Behandlung von bis zu 126 bad sectors ausgelegt. Dabei wird auch auf die Grobeinteilung einer Platte in Spuren und Zylinder Bezug genommen.
Das bad sector file liegt z. B. immer auf der letzten Spur des letzten Zylinders. Ebenfalls umfasst der Ersatzbereich immer mehrere komplette Spuren. Da bei verschiedenen Plattentypen die Spurgrösse (Anzahl Sektoren pro Spur) variiert, sind die Grössen des Ersatzbereiches und bad sector files abhängig vom Plattentyp. Die Aufteilung bei den zur Zeit unterstützten Plattentypen ist der folgenden Tabelle zu entnehmen

Plattentyp	Anzahl der Sektoren			
	gesamte Platte	öffentlicher Bereich	Ersatz-bereich	bad sector file
RL01	20480	20280	160	40
RL02	40960	40760	160	40
RK06	27126	26972	132	22
RK07	53790	53636	132	22
RM02/03	131680	131520	128	32
RM05	500384	500224	128	32

2.1.2. Bad Sector File

Das bad sector file besitzt folgenden Aufbau:

Sektor	Inhalt
0 1	bad sector Information
2 3	nicht benützt
4 5	Kopie von 0,1
6 7	nicht benützt
8 9	Kopie von 0,1
10 11	nicht benützt
12 13	Kopie von 0,1
14 15	nicht benützt
16 17	Kopie von 0,1
18 19	nicht benützt

Die bad sector Information besteht aus 256 16-Bit-Worten:

Wort	Inhalt	
0	0	
1	0	
2	Anzahl bad sectors	
3	0	
4 5	Sektornummer	1. bad sector Eintrag
6 7	Sektornummer	2. bad sector Eintrag
252 253	Sektornummer	125. bad sector Eintrag
254 255	Sektornummer	126. bad sector Eintrag

Die Tabelle der bad sector - Sektornummern heisst *bad sector table*. Unbenutzte bad sector Einträge werden mit "-1" aufgefüllt. Die bad sector table wird in fünffacher Kopie auf einer Platte abgespeichert. Eine Platte ist dann *verwendbar*, wenn eine dieser Kopien *bad-sector-frei* ist und die Gesamtzahl der bad sectors *126* nicht übersteigt.

2.1.3. Zuordnung: Bad Sector – Ersatzsektor

Die Zuordnung bad sector – Ersatzsektor ergibt sich aus der Reihenfolge der Einträge in der bad sector table:

Dem ersten erfassten bad sector (1. bad sector Eintrag) wird der letzte Sektor im Ersatzbereich zugeordnet *(Sektornummer r-1)* dem 2. der vorletzte Sektor im Ersatzbereich *(r-2)*, usw.

2.1.4. Zuordnung: Sektornummer – MUNIX-Blocknummer

MUNIX-Blöcke sind 1024 bytes gross, Plattensektoren 256 (bei RL01/02) oder 512 bytes (RK06/07, RM02/03/05). Ein MUNIX-Block umfasst somit 4 bzw. 2 Sektoren.

Die Standalone-Treiber betrachten eine Platte als lineare Folge von Blöcken mit den Nummern *0* bis *n/4-1* bzw. *n/2-1* (*n* = Anzahl der Sektoren einer Platte). Die Zuordnung lautet (abhängig von der Sektorgrösse):

Sektorgrösse	Blocknummer	Sektornummer(n)
256 bytes	a	4*a – 4*a+3
512 bytes	a	2*a – 2*a+1

Unter MUNIX kann eine Platte in mehrere Filesysteme unterteilt werden. Die Blöcke eines Filesystems werden relativ zum ersten Block des Filesystems durchnumeriert (beginnend mit 0). Die absolute Blocknummer **a** des Blocks eines Filesystems ergibt sich aus der Summation der Blöcke der vorhergehenden Filesysteme (offset) plus der relativen Blocknummer **r**. Aus der absoluten Blocknummer **a** erhält man die zugeordneten Sektornummern über die vorstehende Tabelle.

2.2. Disk-Check-Programme

Aufgabe der Disk-Check-Programme ist es

- alle bad sectors auf einer Platte zu finden
- die bad sector table aufzubauen und das bad sector file auf der Platte anzulegen.

Das Auffinden der bad sectors erfolgt durch Testen der Plattensektoren.

2.2.1. Sektortest

Beim *Sektortest* wird folgendermassen vorgegangen:

1) Schreib-/Lese-Zyklus mit anschliessendem Datenvergleich.
   Der Schreib- bzw. Lesezugriff wird nicht wiederholt, wenn vom Plattencontroller ein Fehler angezeigt wird. Ein Sektor wird als bad betrachtet bei einem Schreib-/Lesefehler oder falls der Vergleich Unterschiede erbringt. Falls möglich, erfolgt eine entsprechende Markierung im Header des Sektors.
2) Wahlweise ist ein Sektortest im *Read-Only-Modus* möglich, um den Inhalt einer Platte nicht zu zerstören. In diesem Fall wird nach einem Lesefehler der Sektor als *bad* betrachtet.

Der Aufbau der bad sector table erfolgt ungeordnet. Neue bad sectors werden ans Ende der Tabelle angefügt. Da sich aus der Reihenfolge der Einträge die Zuordnung zu den Ersatzsektoren ergibt, kann auf diese Weise auch nachträglich ein neuer bad sector in die Tabelle aufgenommen werden. An der Reihenfolge der "alten" bad sectors ändert sich dadurch nichts.

2.2.2. Testabläufe

Die Disk-Check-Programme stellen verschiedene Funktionen zur Verfügung:

1) Vollständiger Test *(bad sector scan)*:
   Die komplette Platte wird getestet. Die Sektoren werden zuerst in aufsteigender Ordnung mit dem DEC-Testmuster **0xeb6db6db** beschrieben. Anschliessend werden die Sektoren in absteigender Ordnung gelesen. Auf die Sektoren wird dabei in grösseren Einheiten zugegriffen (komplette Spur). Tritt dabei ein Fehler auf, wird der Zugriff für jeden Sektor einzeln wiederholt und der bad sector festgestellt. Dabei wird das bad sector file aufgebaut und nach Abschluss des Tests auf die Platte geschrieben.

Die Funktionen 2) bis 4) setzen voraus, dass auf der Platte bereits ein bad sector file existiert.

2) Partieller Test:
   Eine zusammenhängende Folge von Sektoren (im Dialog spezifiziert) wird getestet. Vor diesem Test wird das bad sector file gelesen. Die Sektoren werden alternierend *(1. Sektor, letzter Sektor, 2. Sektor, ...)* mit einem Random-Muster getestet. Beim Test werden eventuell neue bad sectors eingetragen und anschliessend das bad sector file zurückgeschrieben.

3) Manuelles Eintragen von bad sectors:
   Bad sectors, die erst nach längerer Benützung einer Platte entstehen, werden von den Disk-Check-Programmen nicht in allen Fällen registriert. Diese Funktion dient zur Kennzeichnung solcher bad sectors.
   Das bad sector file wird gelesen. Der Benutzer gibt die Nummern von bad sectors ein. Diese Sektoren werden als *bad* markiert und in die bad sector table aufgenommen. Anschliessend wird das bad sector file zurückgeschrieben.

4) Auflisten aller bekannten bad sectors:
   Das bad sector file wird eingelesen. Die Nummern aller bad sectors werden aufgelistet.

Für einen Langzeittest steht eine weitere Funktion zur Verfügung:

5) Random-Sektortest:
   Ein Zufallsgenerator erzeugt eine Folge von Sektornummern. Diese Sektoren werden mit einem Random-Muster getestet.
   Der Test ist nicht terminiert. Er kann durch einen Bus-Reset abgebrochen werden.
   Ein vorhandenes bad sector file auf der Platte wird eventuell zerstört.

2.3. Plattentreiber

Die generelle Vorgehensweise ist beim Standalone-Treiber und beim MUNIX-Treiber ähnlich:

Es werden solange Schreib-/Lesezugriffe auf die Platte ausgeführt, bis auf einen bad sector zugegriffen wird. (Die bad sector Erkennung kann vom Plattentyp abhängig sein; sie wird weiter unten skizziert.)
Falls die bad sector table noch nicht von der Platte gelesen wurde (Anfangszustand bzw. nach Plattenwechsel), wird sie nun eingelesen. Ist der Sektor als bad markiert, wird auf den zugeordneten Ersatzsektor zugegriffen (rekursiv!). Andernfalls erfolgt eine Meldung auf die Systemkonsole: **disk error**.

Erkennen von bad sectors:

a) Der Sektor ist im Header als bad markiert (z.B. RK06/07, RM02/03/05):
   Der Plattencontroller erzeugt eine Fehlermeldung bei einer Schreib-/Leseoperation. Jeder Versuch auf einen als bad markierten Sektor zuzugreifen wird vom Controller mit einer Fehlermeldung quittiert, auf die der Treiber mit einem Zugriff auf den Ersatzsektor reagiert.

b) Der Sektor kann nicht im Header markiert werden (z.B. RL01/02):
   Um zu verhindern, dass ein Sektor zwar fehlerfrei beschrieben, jedoch nicht fehlerfrei gelesen werden kann, wird ein Zugriff auf bad sectors von vorneherein ausgeschlossen. Vor einer Schreib-/Leseoperation wird die bad sector table inspiziert, ob der betroffene Sektor als bad bekannt ist. In diesem Falle wird sofort auf den Ersatzsektor zugegriffen.
   Diese Vorgehensweise setzt voraus, dass vor dem ersten Plattenzugriff bzw. nach einem Plattenwechsel die bad sector table eingelesen wird.

Die Standalone-Treiber unterscheiden sich von den MUNIX-Treibern dadurch, dass generell als erster Plattenzugriff die bad sector table eingelesen wird. Nach einem Plattenwechsel ist deshalb ein Standalone-Programm neu zu starten. Unter MUNIX können dagegen beliebige Plattenwechsel erfolgen, ohne das System neu zu booten.

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

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

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

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

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

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

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

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

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

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

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

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

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

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

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

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

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

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

.MC [column width [gutter width]]

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

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

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

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

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

produces

1. Care and Feeding of Department Heads

Alternatively,

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

will print the heading with no number added:

Care and Feeding of Directors

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

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

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

generates:

2. Erie-Lackawanna

2.1. Morris and Essex Division

2.1.1. Gladstone Branch

2.1.2. Montclair Branch

2.2. Boonton Line

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

```
.NH 0
Penn Central
```

1. Penn Central

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

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

produces

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

[2] Text for second paragraph, ...

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

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

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

will produce

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

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

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

produces this:

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

second: And so forth.

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

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

will result in

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

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

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

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

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

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

```
.I word
```

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

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

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

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

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

```
.UL word
```

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

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

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

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

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

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

whereas

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

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

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

```
.BX word
```

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

Longer pieces of text may be boxed by enclosing them with .B1 and .B2:

```
.B1
text...
.B2
```

as has been done here.

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

* Like this.

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

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

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

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

.ND May 8, 1945

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

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

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

.nr PS 9

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

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

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

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

Input	Output	Input	Output
*'e	é	*_a	ā
*.e	ė	*Ce	ě
*:u	ü	*,c	c
*^e	ê		

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

nroff −ms file

and you can print it on the typesetter with the command

troff −ms file

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

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

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

.EQ L (1.3a)

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

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

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

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

References

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

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

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

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

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

Appendix A
List of Commands

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

Register Names

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

Number registers used in −ms

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

String registers used in −ms

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

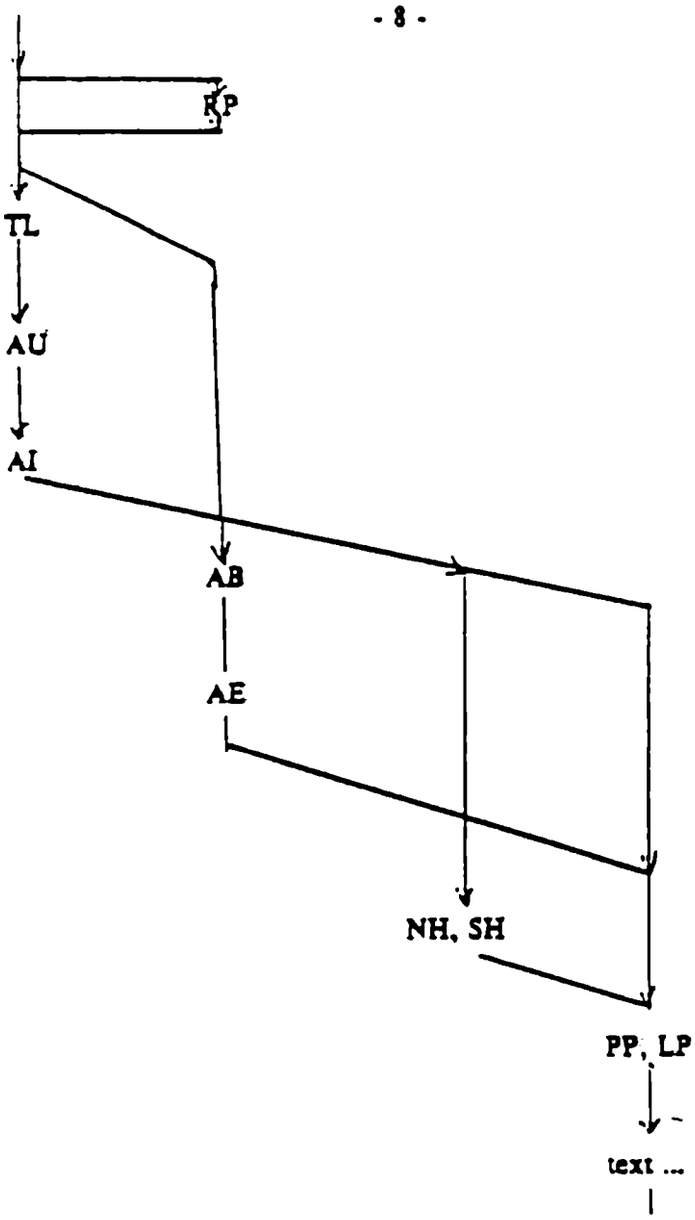


Figure 1

A Guide to Preparing Documents with −ms

M. E. Lesk

Bell Laboratories August 1978

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

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

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

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

Contents

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

UNIX Document no. 1111

Commands for a TM

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

Commands not needed in a particular format are ignored.

Bell Laboratories Cover Sheet for TM

This information is for employees of Bell Laboratories. (GEI 13.9-1)

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

TM- 1978-5b3

Other Keywords- Tools
Design

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

ABSTRACT

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

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

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

A Released Paper with Mathematics

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

... (as for a TM)

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

The Role of the Allen Wrench
in Modern Electronics

J. Q. Principlepusher

X. Y. Hardwired

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

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

April 1, 1976

The Role of the Allen Wrench
in Modern Electronics

J. Q. Principlepusher

X. Y. Hardwired

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

The solution to the torque handle equation

$$\sum_{0}^{\infty} F(x_i) = G(x) \qquad (1)$$

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

An Internal Memorandum

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

Bell Laboratories

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

from: Able, Baker &
Charley, Attys.

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

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

Ordered, adjudged and decreed as follows:

I. [Sherman Act]

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

II. [Definitions]

For the purposes of this Final Judgment:

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

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

Headings

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

1. Introduction
   text text text
```

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

Appendix I
   text text text
```

A Simple List

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

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

Displays

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

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

and now
for something
completely different

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

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

Footnotes

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

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

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

Multiple Indents

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

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

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

Keeps

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

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

Double Column

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

The Declaration of Independence

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

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

Equations

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

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

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

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

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

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

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

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

$$= \left|\frac{\partial V}{\partial x}\right|^2 + \left|\frac{\partial V}{\partial y}\right|^2 \qquad \lambda \longrightarrow \infty$$

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

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

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

Some Registers You Can Change

Line length
.nr LL 7i

Title length
.nr LT 7i

Point size
.nr PS 9

Vertical spacing
.nr VS 11

Column width
.nr CW 3i

Intercolumn spacing
.nr GW .5i

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

Paragraph indent
.nr PI 2n

Paragraph spacing
.nr PD 0

Page offset
.nr PO 0.5i

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

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

Page numbers
.nr % 3

Tables

```
.TS          (Φ indicates a tab)
allbox;
c s s
c c c
n n n.
AT&T Common Stock
Year ΦPrice ΦDividend
1971 Φ41-54 ΦS2.60
2 Φ41-54 Φ2.70
3 Φ46-55 Φ2.87
4 Φ40-53 Φ3.24
5 Φ45-52 Φ3.40
6 Φ51-59 Φ.95*
.TE
* (first quarter only)
```

AT&T Common Stock		
Year	Price	Dividend
1971	41-54	S2.60
2	41-54	2.70
3	46-55	2.87
4	40-53	3.24
5	45-52	3.40
6	51-59	.95*

* (first quarter only)

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

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

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

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

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

Usage

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

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

The UNIX Time-Sharing System

Trademarks:

MUNIX, CADMUS	for PCS
DEC, PDP	for DEC
UNIX	for Bell Laboratories

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."[3]

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, arg1, arg2, ... , argn )
```

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, **wait**s 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 Implementation

UNIX Implementation

K. Thompson

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes in high-level terms the implementation of the resident UNIX† kernel. This discussion is broken into three parts. The first part describes how the UNIX system views processes, users, and programs. The second part describes the I/O system. The last part describes the UNIX file system.

1. INTRODUCTION

The UNIX kernel consists of about 10,000 lines of C code and about 1,000 lines of assembly code. The assembly code can be further broken down into 200 lines included for the sake of efficiency (they could have been written in C) and 800 lines to perform hardware functions not possible in C.

This code represents 5 to 10 percent of what has been lumped into the broad expression "the UNIX operating system." The kernel is the only UNIX code that cannot be substituted by a user to his own liking. For this reason, the kernel should make as few real decisions as possible. This does not mean to allow the user a million options to do the same thing. Rather, it means to allow only one way to do one thing, but have that way be the least-common divisor of all the options that might have been provided.

What is or is not implemented in the kernel represents both a great responsibility and a great power. It is a soap-box platform on "the way things should be done." Even so, if "the way" is too radical, no one will follow it. Every important decision was weighed carefully. Throughout, simplicity has been substituted for efficiency. Complex algorithms are used only if their complexity can be localized.

2. PROCESS CONTROL

In the UNIX system, a user executes programs in an environment called a user process. When a system function is required, the user process calls the system as a subroutine. At some point in this call, there is a distinct switch of environments. After this, the process is said to be a system process. In the normal definition of processes, the user and system processes are different phases of the same process (they never execute simultaneously). For protection, each system process has its own stack.

The user process may execute from a read-only text segment, which is shared by all processes executing the same code. There is no *functional* benefit from shared-text segments. An *efficiency* benefit comes from the fact that there is no need to swap read-only segments out because the original copy on secondary memory is still current. This is a great benefit to interactive programs that tend to be swapped while waiting for terminal input. Furthermore, if two processes are executing simultaneously from the same copy of a read-only segment, only one copy needs to reside in primary memory. This is a secondary effect, because simultaneous

†UNIX is a Trademark of Bell Laboratories.

execution of a program is not common. It is ironic that this effect, which reduces the use of primary memory, only comes into play when there is an overabundance of primary memory, that is, when there is enough memory to keep waiting processes loaded.

All current read-only text segments in the system are maintained from the *text table*. A text table entry holds the location of the text segment on secondary memory. If the segment is loaded, that table also holds the primary memory location and the count of the number of processes sharing this entry. When this count is reduced to zero, the entry is freed along with any primary and secondary memory holding the segment. When a process first executes a shared-text segment, a text table entry is allocated and the segment is loaded onto secondary memory. If a second process executes a text segment that is already allocated, the entry reference count is simply incremented.

A user process has some strictly private read-write data contained in its data segment. As far as possible, the system does not use the user's data segment to hold system data. In particular, there are no I/O buffers in the user address space.

The user data segment has two growing boundaries. One, increased automatically by the system as a result of memory faults, is used for a stack. The second boundary is only grown (or shrunk) by explicit requests. The contents of newly allocated primary memory is initialized to zero.

Also associated and swapped with a process is a small fixed-size system data segment. This segment contains all the data about the process that the system needs only when the process is active. Examples of the kind of data contained in the system data segment are: saved central processor registers, open file descriptors, accounting information, scratch data area, and the stack for the system phase of the process. The system data segment is not addressable from the user process and is therefore protected.

Last, there is a process table with one entry per process. This entry contains all the data needed by the system when the process is *not* active. Examples are the process's name, the location of the other segments, and scheduling information. The process table entry is allocated when the process is created, and freed when the process terminates. This process entry is always directly addressable by the kernel.

Figure 1 shows the relationships between the various process control data. In a sense, the process table is the definition of all processes, because all the data associated with a process may be accessed starting from the process table entry.

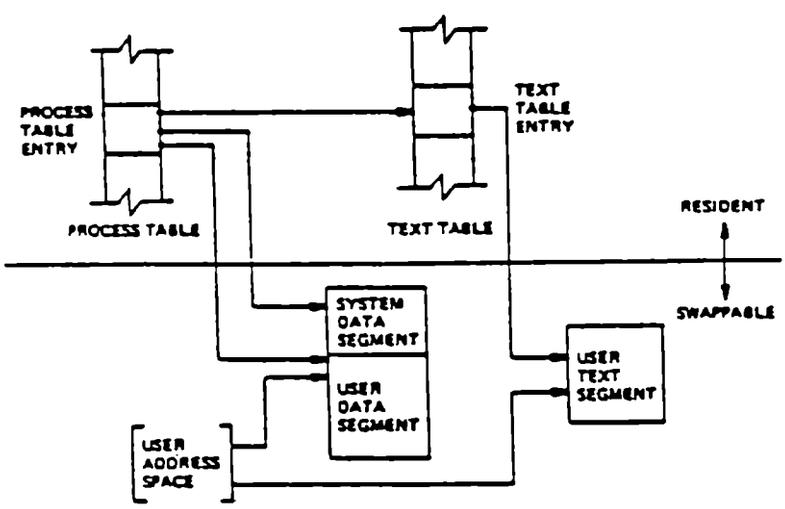


Fig. 1—Process control data structure.

2.1. Process creation and program execution

Processes are created by the system primitive **fork**. The newly created process (child) is a copy of the original process (parent). There is no detectable sharing of primary memory between the two processes. (Of course, if the parent process was executing from a read-only text segment, the child will share the text segment.) Copies of all writable data segments are made for the child process. Files that were open before the **fork** are truly shared after the **fork**. The processes are informed as to their part in the relationship to allow them to select their own (usually non-identical) destiny. The parent may **wait** for the termination of any of its children.

A process may **exec** a file. This consists of exchanging the current text and data segments of the process for new text and data segments specified in the file. The old segments are lost. Doing an **exec** does *not* change processes; the process that did the **exec** persists, but after the **exec** it is executing a different program. Files that were open before the **exec** remain open after the **exec**.

If a program, say the first pass of a compiler, wishes to overlay itself with another program, say the second pass, then it simply **exec**s the second program. This is analogous to a "goto." If a program wishes to regain control after **exec**ing a second program, it should **fork** a child process, have the child **exec** the second program, and have the parent **wait** for the child. This is analogous to a "call." Breaking up the call into a binding followed by a transfer is similar to the subroutine linkage in SL-5.[1]

2.2. Swapping

The major data associated with a process (the user data segment, the system data segment, and the text segment) are swapped to and from secondary memory, as needed. The user data segment and the system data segment are kept in contiguous primary memory to reduce swapping latency. (When low-latency devices, such as bubbles, CCDs, or scatter/gather devices, are used, this decision will have to be reconsidered.) Allocation of both primary and secondary memory is performed by the same simple first-fit algorithm. When a process grows, a new piece of primary memory is allocated. The contents of the old memory is copied to the new memory. The old memory is freed and the tables are updated. If there is not enough primary memory, secondary memory is allocated instead. The process is swapped out onto the secondary memory, ready to be swapped in with its new size.

One separate process in the kernel, the swapping process, simply swaps the other processes in and out of primary memory. It examines the process table looking for a process that is swapped out and is ready to run. It allocates primary memory for that process and reads its segments into primary memory, where that process competes for the central processor with other loaded processes. If no primary memory is available, the swapping process makes memory available by examining the process table for processes that can be swapped out. It selects a process to swap out, writes it to secondary memory, frees the primary memory, and then goes back to look for a process to swap in.

Thus there are two specific algorithms to the swapping process. Which of the possibly many processes that are swapped out is to be swapped in? This is decided by secondary storage residence time. The one with the longest time out is swapped in first. There is a slight penalty for larger processes. Which of the possibly many processes that are loaded is to be swapped out? Processes that are waiting for slow events (i.e., not currently running or waiting for disk I/O) are picked first, by age in primary memory, again with size penalties. The other processes are examined by the same age algorithm, but are not taken out unless they are at least of some age. This adds hysteresis to the swapping and prevents total thrashing.

These swapping algorithms are the most suspect in the system. With limited primary memory, these algorithms cause total swapping. This is not bad in itself, because the swapping does not impact the execution of the resident processes. However, if the swapping device must also be used for file storage, the swapping traffic severely impacts the file system traffic. It is exactly these small systems that tend to double usage of limited disk resources.

2.3. Synchronization and scheduling

Process synchronization is accomplished by having processes wait for events. Events are represented by arbitrary integers. By convention, events are chosen to be addresses of tables associated with those events. For example, a process that is waiting for any of its children to terminate will wait for an event that is the address of its own process table entry. When a process terminates, it signals the event represented by its parent's process table entry. Signaling an event on which no process is waiting has no effect. Similarly, signaling an event on which many processes are waiting will wake all of them up. This differs considerably from Dijkstra's P and V synchronization operations,[2] in that no memory is associated with events. Thus there need be no allocation of events prior to their use. Events exist simply by being used.

On the negative side, because there is no memory associated with events, no notion of "how much" can be signaled via the event mechanism. For example, processes that want memory might wait on an event associated with memory allocation. When any amount of memory becomes available, the event would be signaled. All the competing processes would then wake up to fight over the new memory. (In reality, the swapping process is the only process that waits for primary memory to become available.)

If an event occurs between the time a process decides to wait for that event and the time that process enters the wait state, then the process will wait on an event that has already happened (and may never happen again). This race condition happens because there is no memory associated with the event to indicate that the event has occurred; the only action of an event is to change a set of processes from wait state to run state. This problem is relieved largely by the fact that process switching can only occur in the kernel by explicit calls to the event-wait mechanism. If the event in question is signaled by another process, then there is no problem. But if the event is signaled by a hardware interrupt, then special care must be taken. These synchronization races pose the biggest problem when UNIX is adapted to multiple-processor configurations.[3]

The event-wait code in the kernel is like a co-routine linkage. At any time, all but one of the processes has called event-wait. The remaining process is the one currently executing. When it calls event-wait, a process whose event has been signaled is selected and that process returns from its call to event-wait.

Which of the runable processes is to run next? Associated with each process is a priority. The priority of a system process is assigned by the code issuing the wait on an event. This is roughly equivalent to the response that one would expect on such an event. Disk events have high priority, teletype events are low, and time-of-day events are very low. (From observation, the difference in system process priorities has little or no performance impact.) All user-process priorities are lower than the lowest system priority. User-process priorities are assigned by an algorithm based on the recent ratio of the amount of compute time to real time consumed by the process. A process that has used a lot of compute time in the last real-time unit is assigned a low user priority. Because interactive processes are characterized by low ratios of compute to real time, interactive response is maintained without any special arrangements.

The scheduling algorithm simply picks the process with the highest priority, thus picking all system processes first and user processes second. The compute-to-real-time ratio is updated every second. Thus, all other things being equal, looping user processes will be scheduled round-robin with a 1-second quantum. A high-priority process waking up will preempt a running, low-priority process. The scheduling algorithm has a very desirable negative feedback character. If a process uses its high priority to hog the computer, its priority will drop. At the same time, if a low-priority process is ignored for a long time, its priority will rise.

3. I/O SYSTEM

The I/O system is broken into two completely separate systems: the block I/O system and the character I/O system. In retrospect, the names should have been "structured I/O" and "unstructured I/O," respectively; while the term "block I/O" has some meaning, "character

I/O" is a complete misnomer.

Devices are characterized by a major device number, a minor device number, and a class (block or character). For each class, there is an array of entry points into the device drivers. The major device number is used to index the array when calling the code for a particular device driver. The minor device number is passed to the device driver as an argument. The minor number has no significance other than that attributed to it by the driver. Usually, the driver uses the minor number to access one of several identical physical devices.

The use of the array of entry points (configuration table) as the only connection between the system code and the device drivers is very important. Early versions of the system had a much less formal connection with the drivers, so that it was extremely hard to handcraft differently configured systems. Now it is possible to create new device drivers in an average of a few hours. The configuration table in most cases is created automatically by a program that reads the system's parts list.

3.1. Block I/O system

The model block I/O device consists of randomly addressed, secondary memory blocks of 512 bytes each. The blocks are uniformly addressed 0, 1, ... up to the size of the device. The block device driver has the job of emulating this model on a physical device.

The block I/O devices are accessed through a layer of buffering software. The system maintains a list of buffers (typically between 10 and 70) each assigned a device name and a device address. This buffer pool constitutes a data cache for the block devices. On a read request, the cache is searched for the desired block. If the block is found, the data are made available to the requester without any physical I/O. If the block is not in the cache, the least recently used block in the cache is renamed, the correct device driver is called to fill up the renamed buffer, and then the data are made available. Write requests are handled in an analogous manner. The correct buffer is found and relabeled if necessary. The write is performed simply by marking the buffer as "dirty." The physical I/O is then deferred until the buffer is renamed.

The benefits in reduction of physical I/O of this scheme are substantial, especially considering the file system implementation. There are, however, some drawbacks. The asynchronous nature of the algorithm makes error reporting and meaningful user error handling almost impossible. The cavalier approach to I/O error handling in the UNIX system is partly due to the asynchronous nature of the block I/O system. A second problem is in the delayed writes. If the system stops unexpectedly, it is almost certain that there is a lot of logically complete, but physically incomplete, I/O in the buffers. There is a system primitive to flush all outstanding I/O activity from the buffers. Periodic use of this primitive helps, but does not solve, the problem. Finally, the associativity in the buffers can alter the physical I/O sequence from that of the logical I/O sequence. This means that there are times when data structures on disk are inconsistent, even though the software is careful to perform I/O in the correct order. On nonrandom devices, notably magnetic tape, the inversions of writes can be disastrous. The problem with magnetic tapes is "cured" by allowing only one outstanding write request per drive.

3.2. Character I/O system

The character I/O system consists of all devices that do not fall into the block I/O model. This includes the "classical" character devices such as communications lines, paper tape, and line printers. It also includes magnetic tape and disks when they are not used in a stereotyped way, for example, 80-byte physical records on tape and track-at-a-time disk copies. In short, the character I/O interface means "everything other than block." I/O requests from the user are sent to the device driver essentially unaltered. The implementation of these requests is, of course, up to the device driver. There are guidelines and conventions to help the implementation of certain types of device drivers.

3.2.1. Disk drivers

Disk drivers are implemented with a queue of transaction records. Each record holds a read/write flag, a primary memory address, a secondary memory address, and a transfer byte count. Swapping is accomplished by passing such a record to the swapping device driver. The block I/O interface is implemented by passing such records with requests to fill and empty system buffers. The character I/O interface to the disk drivers create a transaction record that points directly into the user area. The routine that creates this record also insures that the user is not swapped during this I/O transaction. Thus by implementing the general disk driver, it is possible to use the disk as a block device, a character device; and a swap device. The only really disk-specific code in normal disk drivers is the pre-sort of transactions to minimize latency for a particular device, and the actual issuing of the I/O request.

3.2.2. Character lists

Real character-oriented devices may be implemented using the common code to handle character lists. A character list is a queue of characters. One routine puts a character on a queue. Another gets a character from a queue. It is also possible to ask how many characters are currently on a queue. Storage for all queues in the system comes from a single common pool. Putting a character on a queue will allocate space from the common pool and link the character onto the data structure defining the queue. Getting a character from a queue returns the corresponding space to the pool.

A typical character-output device (paper tape punch, for example) is implemented by passing characters from the user onto a character queue until some maximum number of characters is on the queue. The I/O is prodded to start as soon as there is anything on the queue and, once started, it is sustained by hardware completion interrupts. Each time there is a completion interrupt, the driver gets the next character from the queue and sends it to the hardware. The number of characters on the queue is checked and, as the count falls through some intermediate level, an event (the queue address) is signaled. The process that is passing characters from the user to the queue can be waiting on the event, and refill the queue to its maximum when the event occurs.

A typical character input device (for example, a paper tape reader) is handled in a very similar manner.

Another class of character devices is the terminals. A terminal is represented by three character queues. There are two input queues (raw and canonical) and an output queue. Characters going to the output of a terminal are handled by common code exactly as described above. The main difference is that there is also code to interpret the output stream as ASCII characters and to perform some translations, e.g., escapes for deficient terminals. Another common aspect of terminals is code to insert real-time delay after certain control characters.

Input on terminals is a little different. Characters are collected from the terminal and placed on a raw input queue. Some device-dependent code conversion and escape interpretation is handled here. When a line is complete in the raw queue, an event is signaled. The code catching this signal then copies a line from the raw queue to a canonical queue performing the character erase and line kill editing. User read requests on terminals can be directed at either the raw or canonical queues.

3.2.3. Other character devices

Finally, there are devices that fit no general category. These devices are set up as character I/O drivers. An example is a driver that reads and writes unmapped primary memory as an I/O device. Some devices are too fast to be treated a character at time, but do not fit the disk I/O mold. Examples are fast communications lines and fast line printers. These devices either have their own buffers or "borrow" block I/O buffers for a while and then give them back.

4. THE FILE SYSTEM

In the UNIX system, a file is a (one-dimensional) array of bytes. No other structure of files is implied by the system. Files are attached anywhere (and possibly multiply) onto a hierarchy of directories. Directories are simply files that users cannot write. For a further discussion of the external view of files and directories, see Ref. 4.

The UNIX file system is a disk data structure accessed completely through the block I/O system. As stated before, the canonical view of a "disk" is a randomly addressable array of 512-byte blocks. A file system breaks the disk into four self-identifying regions. The first block (address 0) is unused by the file system. It is left aside for booting procedures. The second block (address 1) contains the so-called "super-block." This block, among other things, contains the size of the disk and the boundaries of the other regions. Next comes the i-list, a list of file definitions. Each file definition is a 64-byte structure, called an i-node. The offset of a particular i-node within the i-list is called its i-number. The combination of device name (major and minor numbers) and i-number serves to uniquely name a particular file. After the i-list, and to the end of the disk, come free storage blocks that are available for the contents of files.

The free space on a disk is maintained by a linked list of available disk blocks. Every block in this chain contains a disk address of the next block in the chain. The remaining space contains the address of up to 50 disk blocks that are also free. Thus with one I/O operation, the system obtains 50 free blocks and a pointer where to find more. The disk allocation algorithms are very straightforward. Since all allocation is in fixed-size blocks and there is strict accounting of space, there is no need to compact or garbage collect. However, as disk space becomes dispersed, latency gradually increases. Some installations choose to occasionally compact disk space to reduce latency.

An i-node contains 13 disk addresses. The first 10 of these addresses point directly at the first 10 blocks of a file. If a file is larger than 10 blocks (5,120 bytes), then the eleventh address points at a block that contains the addresses of the next 128 blocks of the file. If the file is still larger than this (70,656 bytes), then the twelfth block points at up to 128 blocks, each pointing to 128 blocks of the file. Files yet larger (8,459,264 bytes) use the thirteenth address for a "triple indirect" address. The algorithm ends here with the maximum file size of 1,082,201,087 bytes.

A logical directory hierarchy is added to this flat physical structure simply by adding a new type of file, the directory. A directory is accessed exactly as an ordinary file. It contains 16-byte entries consisting of a 14-byte name and an i-number. The root of the hierarchy is at a known i-number (*viz.*, 2). The file system structure allows an arbitrary, directed graph of directories with regular files linked in at arbitrary places in this graph. In fact, very early UNIX systems used such a structure. Administration of such a structure became so chaotic that later systems were restricted to a directory tree. Even now, with regular files linked multiply into arbitrary places in the tree, accounting for space has become a problem. It may become necessary to restrict the entire structure to a tree, and allow a new form of linking that is subservient to the tree structure.

The file system allows easy creation, easy removal, easy random accessing, and very easy space allocation. With most physical addresses confined to a small contiguous section of disk, it is also easy to dump, restore, and check the consistency of the file system. Large files suffer from indirect addressing, but the cache prevents most of the implied physical I/O without adding much execution. The space overhead properties of this scheme are quite good. For example, on one particular file system, there are 25,000 files containing 130M bytes of data-file content. The overhead (i-node, indirect blocks, and last block breakage) is about 11.5M bytes. The directory structure to support these files has about 1,500 directories containing 0.6M bytes of directory content and about 0.5M bytes of overhead in accessing the directories. Added up any way, this comes out to less than a 10 percent overhead for actual stored data. Most systems have this much overhead in padded trailing blanks alone.

4.1. File system implementation

Because the i-node defines a file, the implementation of the file system centers around access to the i-node. The system maintains a table of all active i-nodes. As a new file is accessed, the system locates the corresponding i-node, allocates an i-node table entry, and reads the i-node into primary memory. As in the buffer cache, the table entry is considered to be the current version of the i-node. Modifications to the i-node are made to the table entry. When the last access to the i-node goes away, the table entry is copied back to the secondary store i-list and the table entry is freed.

All I/O operations on files are carried out with the aid of the corresponding i-node table entry. The accessing of a file is a straightforward implementation of the algorithms mentioned previously. The user is not aware of i-nodes and i-numbers. References to the file system are made in terms of path names of the directory tree. Converting a path name into an i-node table entry is also straightforward. Starting at some known i-node (the root or the current directory of some process), the next component of the path name is searched by reading the directory. This gives an i-number and an implied device (that of the directory). Thus the next i-node table entry can be accessed. If that was the last component of the path name, then this i-node is the result. If not, this i-node is the directory needed to look up the next component of the path name, and the algorithm is repeated.

The user process accesses the file system with certain primitives. The most common of these are **open**, **create**, **read**, **write**, **seek**, and **close**. The data structures maintained are shown in Fig. 2.

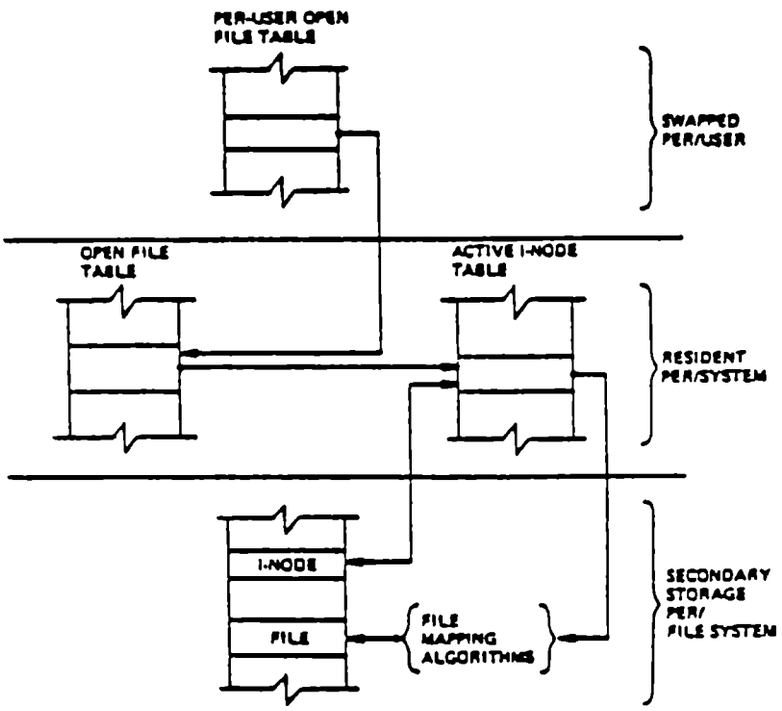


Fig. 2—File system data structure.

In the system data segment associated with a user, there is room for some (usually between 10 and 50) open files. This open file table consists of pointers that can be used to access corresponding i-node table entries. Associated with each of these open files is a current I/O pointer. This is a byte offset of the next read/write operation on the file. The system treats each read/write request as random with an implied seek to the I/O pointer. The user usually thinks of the file as sequential with the I/O pointer automatically counting the number of bytes that have been read/written from the file. The user may, of course, perform random I/O by setting the I/O pointer before reads/writes.

With file sharing, it is necessary to allow related processes to share a common I/O pointer

and yet have separate I/O pointers for independent processes that access the same file. With these two conditions, the I/O pointer cannot reside in the i-node table nor can it reside in the list of open files for the process. A new table (the open file table) was invented for the sole purpose of holding the I/O pointer. Processes that share the same open file (the result of **forks**) share a common open file table entry. A separate open of the same file will only share the i-node table entry, but will have distinct open file table entries.

The main file system primitives are implemented as follows. **open** converts a file system path name into an i-node table entry. A pointer to the i-node table entry is placed in a newly created open file table entry. A pointer to the file table entry is placed in the system data segment for the process. **create** first creates a new i-node entry, writes the i-number into a directory, and then builds the same structure as for an **open**. **read** and **write** just access the i-node entry as described above. **seek** simply manipulates the I/O pointer. No physical seeking is done. **close** just frees the structures built by **open** and **create**. Reference counts are kept on the open file table entries and the i-node table entries to free these structures after the last reference goes away. **unlink** simply decrements the count of the number of directories pointing at the given i-node. When the last reference to an i-node table entry goes away, if the i-node has no directories pointing to it, then the file is removed and the i-node is freed. This delayed removal of files prevents problems arising from removing active files. A file may be removed while still open. The resulting unnamed file vanishes when the file is closed. This is a method of obtaining temporary files.

There is a type of unnamed FIFO file called a **pipe**. Implementation of **pipes** consists of implied **seeks** before each **read** or **write** in order to implement first-in-first-out. There are also checks and synchronization to prevent the writer from grossly outproducing the reader and to prevent the reader from overtaking the writer.

4.2. Mounted file systems

The file system of a UNIX system starts with some designated block device formatted as described above to contain a hierarchy. The root of this structure is the root of the UNIX file system. A second formatted block device may be mounted at any leaf of the current hierarchy. This logically extends the current hierarchy. The implementation of mounting is trivial. A mount table is maintained containing pairs of designated leaf i-nodes and block devices. When converting a path name into an i-node, a check is made to see if the new i-node is a designated leaf. If it is, the i-node of the root of the block device replaces it.

Allocation of space for a file is taken from the free pool on the device on which the file lives. Thus a file system consisting of many mounted devices does not have a common pool of free secondary storage space. This separation of space on different devices is necessary to allow easy unmounting of a device.

4.3. Other system functions

There are some other things that the system does for the user—a little accounting, a little tracing/debugging, and a little access protection. Most of these things are not very well developed because our use of the system in computing science research does not need them. There are some features that are missed in some applications, for example, better inter-process communication.

The UNIX kernel is an I/O multiplexer more than a complete operating system. This is as it should be. Because of this outlook, many features are found in most other operating systems that are missing from the UNIX kernel. For example, the UNIX kernel does not support file access methods, file disposition, file formats, file maximum size, spooling, command language, logical records, physical records, assignment of logical file names, logical file names, more than one character set, an operator's console, an operator, log-in, or log-out. Many of these things are symptoms rather than features. Many of these things are implemented in user software using the kernel as a tool. A good example of this is the command language.[5] Each user may have his own command language. Maintenance of such code is as easy as maintaining user

code. The idea of implementing "system" code with general user primitives comes directly from MULTICS.[6]

References

1. R. E. Griswold and D. R. Hanson, "An Overview of SL5," *SIGPLAN Notices* 12(4) pp. 40-50 (April 1977).
2. E. W. Dijkstra, "Cooperating Sequential Processes," pp. 43-112 in *Programming Languages*, ed. F. Genuys, Academic Press, New York (1968).
3. J. A. Hawley and W. B. Meyer, "MUNIX, A Multiprocessing Version of UNIX," M.S. Thesis, Naval Postgraduate School, Monterey, Cal. (1975).
4. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Bell Sys. Tech. J.* 57(6) pp. 1905-1929 (1978).
5. S. R. Bourne, "UNIX Time-Sharing System: The UNIX Shell," *Bell Sys. Tech. J.* 57(6) pp. 1971-1990 (1978).
6. E. I. Organick, *The MULTICS System*, M.I.T. Press, Cambridge, Mass. (1972).

The UNIX I/O System

Trademarks:

MUNIX, CADMUS	for PCS
DEC, PDP	for DEC
UNIX	for Bell Laboratories

The UNIX I/O System

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

This paper gives an overview of the workings of the UNIX† I/O system. It was written with an eye toward providing guidance to writers of device driver routines, and is oriented more toward describing the environment and nature of device drivers than the implementation of that part of the file system which deals with ordinary files.

It is assumed that the reader has a good knowledge of the overall structure of the file system as discussed in the paper "The UNIX Time-sharing System." A more detailed discussion appears in "UNIX Implementation;" the current document restates parts of that one, but is still more detailed. It is most useful in conjunction with a copy of the system code, since it is basically an exegesis of that code.

Device Classes

There are two classes of device: *block* and *character*. The block interface is suitable for devices like disks, tapes, and DECtape which work, or can work, with addressible 512-byte blocks. Ordinary magnetic tape just barely fits in this category, since by use of forward and backward spacing any block can be read, even though blocks can be written only at the end of the tape. Block devices can at least potentially contain a mounted file system. The interface to block devices is very highly structured; the drivers for these devices share a great many routines as well as a pool of buffers.

Character-type devices have a much more straightforward interface, although more work must be done by the driver itself.

Devices of both types are named by a *major* and a *minor* device number. These numbers are generally stored as an integer with the minor device number in the low-order 8 bits and the major device number in the next-higher 8 bits; macros *major* and *minor* are available to access these numbers. The major device number selects which driver will deal with the device; the minor device number is not used by the rest of the system but is passed to the driver at appropriate times. Typically the minor number selects a subdevice attached to a given controller, or one of several similar hardware interfaces.

The major device numbers for block and character devices are used as indices in separate tables; they both start at 0 and therefore overlap.

Overview of I/O

The purpose of the *open* and *creat* system calls is to set up entries in three separate system tables. The first of these is the *u_ofile* table, which is stored in the system's per-process data area *u*. This table is indexed by the file descriptor returned by the *open* or *creat*, and is accessed during a *read*, *write*, or other operation on the open file. An entry contains only a pointer to the corresponding entry of the *file* table, which is a per-system data base. There is one entry in the *file* table for each instance of *open* or *creat*. This table is per-system because the same instance of an open file must be shared among the several processes which can result from *forks* after

†UNIX is a Trademark of Bell Laboratories.

the file is opened. A *file* table entry contains flags which indicate whether the file was open for reading or writing or is a pipe, and a count which is used to decide when all processes using the entry have terminated or closed the file (so the entry can be abandoned). There is also a 32-bit file offset which is used to indicate where in the file the next read or write will take place. Finally, there is a pointer to the entry for the file in the *inode* table, which contains a copy of the file's i-node.

Certain open files can be designated "multiplexed" files, and several other flags apply to such channels. In such a case, instead of an offset, there is a pointer to an associated multiplex channel table. Multiplex channels will not be discussed here.

An entry in the *file* table corresponds precisely to an instance of *open* or *creat;* if the same file is opened several times, it will have several entries in this table. However, there is at most one entry in the *inode* table for a given file. Also, a file may enter the *inode* table not only because it is open, but also because it is the current directory of some process or because it is a special file containing a currently-mounted file system.

An entry in the *inode* table differs somewhat from the corresponding i-node as stored on the disk; the modified and accessed times are not stored, and the entry is augmented by a flag word containing information about the entry, a count used to determine when it may be allowed to disappear, and the device and i-number whence the entry came. Also, the several block numbers that give addressing information for the file are expanded from the 3-byte, compressed format used on the disk to full *long* quantities.

During the processing of an *open* or *creat* call for a special file, the system always calls the device's *open* routine to allow for any special processing required (rewinding a tape, turning on the data-terminal-ready lead of a modem, etc.). However, the *close* routine is called only when the last process closes a file, that is, when the i-node table entry is being deallocated. Thus it is not feasible for a device to maintain, or depend on, a count of its users, although it is quite possible to implement an exclusive-use device which cannot be reopened until it has been closed.

When a *read* or *write* takes place, the user's arguments and the *file* table entry are used to set up the variables *u.u_base, u.u_count,* and *u.u_offset* which respectively contain the (user) address of the I/O target area, the byte-count for the transfer, and the current location in the file. If the file referred to is a character-type special file, the appropriate read or write routine is called; it is responsible for transferring data and updating the count and current location appropriately as discussed below. Otherwise, the current location is used to calculate a logical block number in the file. If the file is an ordinary file the logical block number must be mapped (possibly using indirect blocks) to a physical block number; a block-type special file need not be mapped. This mapping is performed by the *bmap* routine. In any event, the resulting physical block number is used, as discussed below, to read or write the appropriate device.

Character Device Drivers

The *cdevsw* table specifies the interface routines present for character devices. Each device provides five routines: open, close, read, write, and special-function (to implement the *ioctl* system call). Any of these may be missing. If a call on the routine should be ignored, (e.g. *open* on non-exclusive devices that require no setup) the *cdevsw* entry can be given as *nulldev;* if it should be considered an error, (e.g. *write* on read-only devices) *nodev* is used. For terminals, the *cdevsw* structure also contains a pointer to the *tty* structure associated with the terminal.

The *open* routine is called each time the file is opened with the full device number as argument. The second argument is a flag which is non-zero only if the device is to be written upon.

The *close* routine is called only when the file is closed for the last time, that is when the very last process in which the file is open closes it. This means it is not possible for the driver to maintain its own count of its users. The first argument is the device number; the second is a

flag which is non-zero if the file was open for writing in the process which performs the final *close*.

When *write* is called, it is supplied the device as argument. The per-user variable *u.u_count* has been set to the number of characters indicated by the user; for character devices, this number may be 0 initially. *u.u_base* is the address supplied by the user from which to start taking characters. The system may call the routine internally, so the flag *u.u_segflg* is supplied that indicates, if *on*, that *u.u_base* refers to the system address space instead of the user's.

The *write* routine should copy up to *u.u_count* characters from the user's buffer to the device, decrementing *u.u_count* for each character passed. For most drivers, which work one character at a time, the routine *cpass( )* is used to pick up characters from the user's buffer. Successive calls on it return the characters to be written until *u.u_count* goes to 0 or an error occurs, when it returns −1. *Cpass* takes care of interrogating *u.u_segflg* and updating *u.u_count*.

Write routines which want to transfer a probably large number of characters into an internal buffer may also use the routine *iomove(buffer, offset, count, flag)* which is faster when many characters must be moved. *Iomove* transfers up to *count* characters into the *buffer* starting *offset* bytes from the start of the buffer; *flag* should be *B_WRITE* (which is 0) in the write case. Caution: the caller is responsible for making sure the count is not too large and is non-zero. As an efficiency note, *iomove* is much slower if any of *buffer+offset*, *count* or *u.u_base* is odd.

The device's *read* routine is called under conditions similar to *write*, except that *u.u_count* is guaranteed to be non-zero. To return characters to the user, the routine *passc(c)* is available; it takes care of housekeeping like *cpass* and returns −1 as the last character specified by *u.u_count* is returned to the user; before that time, 0 is returned. *Iomove* is also usable as with *write*; the flag should be *B_READ* but the same cautions apply.

The "special-functions" routine is invoked by the *stty* and *gtty* system calls as follows: *(*p)(dev, v)* where *p* is a pointer to the device's routine, *dev* is the device number, and *v* is a vector. In the *gtty* case, the device is supposed to place up to 3 words of status information into the vector; this will be returned to the caller. In the *stty* case, *v* is 0; the device should take up to 3 words of control information from the array *u.u_arg[0...2]*.

Finally, each device should have appropriate interrupt-time routines. When an interrupt occurs, it is turned into a C-compatible call on the devices's interrupt routine. The interrupt-catching mechanism makes the low-order four bits of the "new PS" word in the trap vector for the interrupt available to the interrupt handler. This is conventionally used by drivers which deal with multiple similar devices to encode the minor device number. After the interrupt has been processed, a return from the interrupt handler will return from the interrupt itself.

A number of subroutines are available which are useful to character device drivers. Most of these handlers, for example, need a place to buffer characters in the internal interface between their "top half" (read/write) and "bottom half" (interrupt) routines. For relatively low data-rate devices, the best mechanism is the character queue maintained by the routines *getc* and *putc*. A queue header has the structure

```
struct {
        int     c_cc;   /* character count */
        char    *c_cf;  /* first character */
        char    *c_cl;  /* last character */
} queue;
```

A character is placed on the end of a queue by *putc(c, &queue)* where *c* is the character and *queue* is the queue header. The routine returns −1 if there is no space to put the character, 0 otherwise. The first character on the queue may be retrieved by *getc(&queue)* which returns either the (non-negative) character or −1 if the queue is empty.

Notice that the space for characters in queues is shared among all devices in the system and in the standard system there are only some 600 character slots available. Thus device handlers, especially write routines, must take care to avoid gobbling up excessive numbers of

characters.

The other major help available to device handlers is the sleep-wakeup mechanism. The call *sleep(event, priority)* causes the process to wait (allowing other processes to run) until the *event* occurs; at that time, the process is marked ready-to-run and the call will return when there is no process with higher *priority*.

The call *wakeup(event)* indicates that the *event* has happened, that is, causes processes sleeping on the event to be awakened. The *event* is an arbitrary quantity agreed upon by the sleeper and the waker-up. By convention, it is the address of some data area used by the driver, which guarantees that events are unique.

Processes sleeping on an event should not assume that the event has really happened; they should check that the conditions which caused them to sleep no longer hold.

Priorities can range from 0 to 127; a higher numerical value indicates a less-favored scheduling situation. A distinction is made between processes sleeping at priority less than the parameter *PZERO* and those at numerically larger priorities. The former cannot be interrupted by signals, although it is conceivable that it may be swapped out. Thus it is a bad idea to sleep with priority less than PZERO on an event which might never occur. On the other hand, calls to *sleep* with larger priority may never return if the process is terminated by some signal in the meantime. Incidentally, it is a gross error to call *sleep* in a routine called at interrupt time, since the process which is running is almost certainly not the process which should go to sleep. Likewise, none of the variables in the user area "*u.*" should be touched, let alone changed, by an interrupt routine.

If a device driver wishes to wait for some event for which it is inconvenient or impossible to supply a *wakeup*, (for example, a device going on-line, which does not generally cause an interrupt), the call *sleep(&lbolt, priority)* may be given. *Lbolt* is an external cell whose address is awakened once every 4 seconds by the clock interrupt routine.

The routines *spl4( ), spl5( ), spl6( ), spl7( )* are available to set the processor priority level as indicated to avoid inconvenient interrupts from the device.

If a device needs to know about real-time intervals, then *timeout(func, arg, interval)* will be useful. This routine arranges that after *interval* sixtieths of a second, the *func* will be called with *arg* as argument, in the style *(*func)(arg)*. Timeouts are used, for example, to provide real-time delays after function characters like new-line and tab in typewriter output, and to terminate an attempt to read the 201 Dataphone *dp* if there is no response within a specified number of seconds. Notice that the number of sixtieths of a second is limited to 32767, since it must appear to be positive, and that only a bounded number of timeouts can be going on at once. Also, the specified *func* is called at clock-interrupt time, so it should conform to the requirements of interrupt routines in general.

The Block-device Interface

Handling of block devices is mediated by a collection of routines that manage a set of buffers containing the images of blocks of data on the various devices. The most important purpose of these routines is to assure that several processes that access the same block of the same device in multiprogrammed fashion maintain a consistent view of the data in the block. A secondary but still important purpose is to increase the efficiency of the system by keeping in-core copies of blocks that are being accessed frequently. The main data base for this mechanism is the table of buffers *buf*. Each buffer header contains a pair of pointers *(b_forw, b_back)* which maintain a doubly-linked list of the buffers associated with a particular block device, and a pair of pointers *(av_forw, av_back)* which generally maintain a doubly-linked list of blocks which are "free," that is, eligible to be reallocated for another transaction. Buffers that have I/O in progress or are busy for other purposes do not appear in this list. The buffer header also contains the device and block number to which the buffer refers, and a pointer to the actual storage associated with the buffer. There is a word count which is the negative of the number of words to be transferred to or from the buffer; there is also an error byte and a

residual word count used to communicate information from an I/O routine to its caller. Finally, there is a flag word with bits indicating the status of the buffer. These flags will be discussed below.

Seven routines constitute the most important part of the interface with the rest of the system. Given a device and block number, both *bread* and *getblk* return a pointer to a buffer header for the block; the difference is that *bread* is guaranteed to return a buffer actually containing the current data for the block, while *getblk* returns a buffer which contains the data in the block only if it is already in core (whether it is or not is indicated by the *B_DONE* bit; see below). In either case the buffer, and the corresponding device block, is made "busy," so that other processes referring to it are obliged to wait until it becomes free. *Getblk* is used, for example, when a block is about to be totally rewritten, so that its previous contents are not useful; still, no other process can be allowed to refer to the block until the new data is placed into it.

The *breada* routine is used to implement read-ahead. it is logically similar to *bread*, but takes as an additional argument the number of a block (on the same device) to be read asynchronously after the specifically requested block is available.

Given a pointer to a buffer, the *brelse* routine makes the buffer again available to other processes. It is called, for example, after data has been extracted following a *bread*. There are three subtly-different write routines, all of which take a buffer pointer as argument, and all of which logically release the buffer for use by others and place it on the free list. *Bwrite* puts the buffer on the appropriate device queue, waits for the write to be done, and sets the user's error flag if required. *Bawrite* places the buffer on the device's queue, but does not wait for completion, so that errors cannot be reflected directly to the user. *Bdwrite* does not start any I/O operation at all, but merely marks the buffer so that if it happens to be grabbed from the free list to contain data from some other block, the data in it will first be written out.

Bwrite is used when one wants to be sure that I/O takes place correctly, and that errors are reflected to the proper user; it is used, for example, when updating i-nodes. *Bawrite* is useful when more overlap is desired (because no wait is required for I/O to finish) but when it is reasonably certain that the write is really required. *Bdwrite* is used when there is doubt that the write is needed at the moment. For example, *bdwrite* is called when the last byte of a *write* system call falls short of the end of a block, on the assumption that another *write* will be given soon which will re-use the same block. On the other hand, as the end of a block is passed, *bawrite* is called, since probably the block will not be accessed again soon and one might as well start the writing process as soon as possible.

In any event, notice that the routines *getblk* and *bread* dedicate the given block exclusively to the use of the caller, and make others wait, while one of *brelse*, *bwrite*, *bawrite*, or *bdwrite* must eventually be called to free the block for use by others.

As mentioned, each buffer header contains a flag word which indicates the status of the buffer. Since they provide one important channel for information between the drivers and the block I/O system, it is important to understand these flags. The following names are manifest constants which select the associated flag bits.

B_READ This bit is set when the buffer is handed to the device strategy routine (see below) to indicate a read operation. The symbol *B_WRITE* is defined as 0 and does not define a flag; it is provided as a mnemonic convenience to callers of routines like *swap* which have a separate argument which indicates read or write.

B_DONE This bit is set to 0 when a block is handed to the the device strategy routine and is turned on when the operation completes, whether normally as the result of an error. It is also used as part of the return argument of *getblk* to indicate if 1 that the returned buffer actually contains the data in the requested block.

B_ERROR This bit may be set to 1 when *B_DONE* is set to indicate that an I/O or other error occurred. If it is set the *b_error* byte of the buffer header may contain an error code if it is non-zero. If *b_error* is 0 the nature of the error is not specified. Actually no driver at present sets *b_error;* the latter is provided for a future improvement whereby a more detailed error-reporting scheme may be implemented.

B_BUSY This bit indicates that the buffer header is not on the free list, i.e. is dedicated to someone's exclusive use. The buffer still remains attached to the list of blocks associated with its device, however. When *getblk* (or *bread*, which calls it) searches the buffer list for a given device and finds the requested block with this bit on, it sleeps until the bit clears.

B_PHYS This bit is set for raw I/O transactions that need to allocate the Unibus map on an 11/70.

B_MAP This bit is set on buffers that have the Unibus map allocated, so that the *iodone* routine knows to deallocate the map.

B_WANTED This flag is used in conjunction with the *B_BUSY* bit. Before sleeping as described just above, *getblk* sets this flag. Conversely, when the block is freed and the busy bit goes down (in *brelse)* a *wakeup* is given for the block header whenever *B_WANTED* is on. This strategem avoids the overhead of having to call *wakeup* every time a buffer is freed on the chance that someone might want it.

B_AGE This bit may be set on buffers just before releasing them; if it is on, the buffer is placed at the head of the free list, rather than at the tail. It is a performance heuristic used when the caller judges that the same block will not soon be used again.

B_ASYNC This bit is set by *bawrite* to indicate to the appropriate device driver that the buffer should be released when the write has been finished, usually at interrupt time. The difference between *bwrite* and *bawrite* is that the former starts I/O, waits until it is done, and frees the buffer. The latter merely sets this bit and starts I/O. The bit indicates that *relse* should be called for the buffer on completion.

B_DELWRI This bit is set by *bdwrite* before releasing the buffer. When *getblk*, while searching for a free block, discovers the bit is 1 in a buffer it would otherwise grab, it causes the block to be written out before reusing it.

Block Device Drivers

The *bdevsw* table contains the names of the interface routines and that of a table for each block device.

Just as for character devices, block device drivers may supply an *open* and a *close* routine called respectively on each open and on the final close of the device. Instead of separate read and write routines, each block device driver has a *strategy* routine which is called with a pointer to a buffer header as argument. As discussed, the buffer header contains a read/write flag, the core address, the block number, a (negative) word count, and the major and minor device number. The role of the strategy routine is to carry out the operation as requested by the information in the buffer header. When the transaction is complete the *B_DONE* (and possibly the *B_ERROR)* bits should be set. Then if the *B_ASYNC* bit is set, *brelse* should be called; otherwise, *wakeup*. In cases where the device is capable, under error-free operation, of transferring fewer words than requested, the device's word-count register should be placed in the residual count slot of the buffer header; otherwise, the residual count should be set to 0. This particular mechanism is really for the benefit of the magtape driver; when reading this device records shorter than requested are quite normal, and the user should be told the actual length of the record.

Although the most usual argument to the strategy routines is a genuine buffer header allocated as discussed above, all that is actually required is that the argument be a pointer to a place containing the appropriate information. For example the *swap* routine, which manages movement of core images to and from the swapping device, uses the strategy routine for this

device. Care has to be taken that no extraneous bits get turned on in the flag word.

The device's table specified by *bdevsw* has a byte to contain an active flag and an error count, a pair of links which constitute the head of the chain of buffers for the device *(b_forw, b_back)*, and a first and last pointer for a device queue. Of these things, all are used solely by the device driver itself except for the buffer-chain pointers. Typically the flag encodes the state of the device, and is used at a minimum to indicate that the device is currently engaged in transferring information and no new command should be issued. The error count is useful for counting retries when errors occur. The device queue is used to remember stacked requests; in the simplest case it may be maintained as a first-in first-out list. Since buffers which have been handed over to the strategy routines are never on the list of free buffers, the pointers in the buffer which maintain the free list *(av_forw, av_back)* are also used to contain the pointers which maintain the device queues.

A couple of routines are provided which are useful to block device drivers. *iodone(bp)* arranges that the buffer to which *bp* points be released or awakened, as appropriate, when the strategy module has finished with the buffer, either normally or after an error. (In the latter case the *B_ERROR* bit has presumably been set.)

The routine *geterror(bp)* can be used to examine the error bit in a buffer header and arrange that any error indication found therein is reflected to the user. It may be called only in the non-interrupt part of a driver when I/O has completed *(B_DONE* has been set).

Raw Block-device I/O

A scheme has been set up whereby block device drivers may provide the ability to transfer information directly between the user's core image and the device without the use of buffers and in blocks as large as the caller requests. The method involves setting up a character-type special file corresponding to the raw device and providing *read* and *write* routines which set up what is usually a private, non-shared buffer header with the appropriate information and call the device's strategy routine. If desired, separate *open* and *close* routines may be provided but this is usually unnecessary. A special-function routine might come in handy, especially for magtape.

A great deal of work has to be done to generate the "appropriate information" to put in the argument buffer for the strategy module; the worst part is to map relocated user addresses to physical addresses. Most of this work is done by *physio(strat, bp, dev, rw)* whose arguments are the name of the strategy routine *strat*, the buffer pointer *bp*, the device number *dev*, and a read-write flag *rw* whose value is either *B_READ* or *B_WRITE*. *Physio* makes sure that the user's base address and count are even (because most devices work in words) and that the core area affected is contiguous in physical space; it delays until the buffer is not busy, and makes it busy while the operation is in progress; and it sets up user error return information.

An Introduction to

Display Editing with VI

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 at 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 paragraghs. A small set of operators, like **d** for delete and **c** for change, are combined with the motion commands to form opterations such as delete word or change paragraph, in a simple and natural way. This regularity and the mnemonic assignment of commands to keys makes the editor command set easy to remember and to use.

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

It is also possible to use the command set of *vi* on hardcopy terminals, storage tubes and "glass tty's" using a one line editing window; thus *vi's* command set is available on all terminals. The full command set of the more traditional, line oriented editor *ex* is available within *vi*; it is quite simple to switch between the two modes of editing.

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 /lastSCR searches for the word 'last' at the end of a line.*

---

‡ If you don't have a "^" key on your terminal then there is probably a key labelled "↑"; in any case these characters are one and the same.

‡‡ Version 3 only.

† Not available in all v2 editors due to memory constraints.

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

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

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

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

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

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

2.3. Moving around on the screen

Now try just moving the cursor around on the screen. If your terminal has arrow keys (4 or 5 keys with arrows going in each direction) try them and convince yourself that they work. (On certain terminals using v2 editors, they won't.) If you don't have working arrow keys, you can always use h, j, k, and l. Experienced users of *vi* prefer these keys to arrow keys, because they are usually right underneath their fingers.

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

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

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

2.4. Moving within a line

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

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

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

2.5. Summary

SPACE	advance the cursor one position
^B	backwards to previous page
^D	scrolls down in the file
^E	exposes another line at the bottom (v3)
^F	forward to next page
^G	tell what is going on
^H	backspace the cursor
^N	next line, same column
^P	previous line, same column
^U	scrolls up in the file
^Y	exposes another line at the top (v3)
+	next line, at the beginning
-	previous line, at the beginning
/	scan for a following string forwards
?	scan backwards
B	back a word, ignoring punctuation
G	go to specified line, last default
H	home screen line
M	middle screen line
L	last screen line
W	forward a word, ignoring punctuation
b	back a word
e	end of current word
n	scan for next instance of / or ? pattern
w	word after this word

2.6. View ‡

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

3. Making simple changes

3.1. Inserting

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

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

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

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

‡ Not available in all v2 editors due to memory constraints.

is inserted on the new line.

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

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

While you are inserting new text, you can use the characters you normally use at the system command level (usually ^H or #) to backspace over the last character which you typed, and the character which you use to kill input lines (usually @, ^X, or ^U) to erase the input you have typed on the current line.† The character ^W will erase a whole word and leave you after the space after the previous word; it is useful for quickly backing up in an insert.

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

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

3.2. Making small corrections

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

If the character is incorrect, you can replace it with the correct character by giving the command rc, where c is replaced by the correct character. Finally if the character which is incorrect should be replaced by more than one character, give the command s which substitutes a string of characters, ending with ESC, for it. If there are a small number of characters which are wrong you can precede s with a count of the number of characters to be replaced. Counts are also useful with x to specify the number of characters to be deleted.

3.3. More corrections: operators

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

---

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

Now try db. This deletes a word backwards, namely the preceding word. Try dSPACE. 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 *fx* where *x* is this character. This command finds the next *x* character to the right of the cursor in the current line. Try then hitting a ;, which finds the next instance of the same character. By using the f command and then a sequence of ;'s you can often get to a particular place in a line much faster than with a sequence of word motions or SPACEs. There is also a F command, which is like f, but searches backward. The ; command repeats F also.

When you are operating on the text in a line it is often desirable to deal with the characters up to, but not including, the first instance of a character. Try d*fx* 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:, 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 :wCR. The first exits from the editor, (writing if changes were made), the second writes and stays in the editor.

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

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

5.2. Escaping to a shell

You can get to a shell to execute a single command by giving a *vi* command of the form :!*cmd*CR. The system will run the single command *cmd* and when the command finishes, the editor will ask you to hit a RETURN to continue. When you have finished looking at the output on the screen, you should hit RETURN and the editor will clear the screen and redraw it. You can then continue editing. You can also give another command when it asks you for a RETURN; in this case the screen will not be redrawn.

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

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

5.3. Marking and returning

The command '' returned to the previous place after a motion of the cursor by a command such as /, ? or G. You can also mark lines in the file with single letter tags and return to these marks later by naming the tags. Try marking the current line with the command *mx* 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 slowCR. If your system is sluggish this helps lessen the amount of output coming to your terminal. You can disable this option by :se noslowCR.

The editor can simulate an intelligent terminal on a dumb one. Try giving the command :se redrawCR. 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 noredrawCR.

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

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

/ ? [[ ]]

Thus if you are searching for a particular instance of a common string in a file you can precede

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

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

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

See section 7.8 on *open* mode for another way to use the *vi* command set on slow terminals.

6.2. Options, set, and editor startup files

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

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

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

set *opt*=*val*

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

set *opt*
set no*opt*

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

You can get a list of all options which you have changed by the command :setCR, or the value of a single option by the command :set *opt*?CR. A list of all possible options and their values is generated by :set allCR. Set can be abbreviated se. Multiple options can be placed on one line, e.g. :se ai aw nuCR.

Options set by the set command only last while you stay in the editor. It is common to want to have certain options set whenever you use the editor. This can be accomplished by creating a list of *ex* commands† which are to be run every time you start up *ex*, *edit*, or *vi*. A

† Note that the command 5z. has an entirely different effect, placing line 5 in the center of a new window.
† All commands which start with : are *ex* commands.

typical list includes a set command, and possibly a few map commands (on v3 editors). Since it is advisable to get these commands on one line, they can be separated with the | character, for example:

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

which sets the options *autoindent*, *autowrite*, *terse*, (the set command), makes @ delete a line, (the first map), and makes # delete a character, (the second map). (See section 6.9 for a description of the map command, which only works in version 3.) This string should be placed in the variable EXINIT in your environment. If you use *csh*, put this line in the file *.login* in your home directory:

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

If you use the standard v7 shell, put these lines in the file *.profile* in your home directory:

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

On a version 6 system, the concept of environments is not present. In this case, put the line in the file *.exrc* in your home directory.

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

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

6.3. Recovering lost lines

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

```
"1pu.u.u.
```

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

6.4. Recovering lost files

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

```
% vi -r name
```

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

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

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

```
% vi -r
```

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

For this feature to work, *vi* must be correctly installed by a super user on your system, and the *mail* program must exist to receive mail. The invocation "*vi -r*" will not always list all saved files, but they can be recovered even if they are not listed.

6.5. Continuous text input

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

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

6.6. Features for editing programs

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

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

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

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

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

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

---

* This feature is not available on some v2 editors. In v2 editors where it is available, the break can only occur to the right of the specified boundary instead of to the left.

6.7. Filtering portions of the buffer

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

6.8. Commands for editing LISP†

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

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

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

The editor also has an operator to realign existing lines as though they had been typed in with *lisp* and *autoindent* set. This is the = operator. Try the command =% at the beginning of a function. This will realign all the lines of the function declaration.

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

6.9. Macros‡

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

Briefly, there are two flavors of macros:

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

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

:map *lhs rhs*CR

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

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

:map q :wq^V^VCR CR

which means that whenever you type q, it will be as though you had typed the four characters :wqCR. A ^V's is needed because without it the CR would end the : command, rather than

† The LISP features are not available on some v2 editors due to memory constraints.

‡ The macro feature is available only in version 3 editors.

becoming part of the *map* definition. There are two ^V's because from within *vi*, two ^V's must be typed to get one. The first CR is part of the *rhs*, the second terminates the : command.

Macros can be deleted with

unmap lhs

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

:map ^V^V^I #

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

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

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

:map! ^T ^V␣␣␣␣

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

7. Word Abbreviations ‡‡

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

:ab eecs Electrical Engineering and Computer Sciences

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

7.1. Abbreviations

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

8. Nitty-gritty details

8.1. Line representation in the display

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

The editor only puts full lines on the display; if there is not enough room on the display to fit a logical line, the editor leaves the physical line empty, placing only an @ on the line as a

‡‡ Version 3 only.

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

place holder. When you delete lines on a dumb terminal, the editor will often just clear the lines to @ to save time (rather than rewriting the rest of the screen.) You can always maximize the information on the screen by giving the ^R command.

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

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

8.2. Counts

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

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

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

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

The scroll commands ^D and ^U likewise remember the amount of scroll last specified, using half the basic window size initially. The simple insert commands use a count to specify a repetition of the inserted text. Thus 10a+----ESC will insert a grid-like string of text. A few commands also use a preceding count as a line or column number.

Except for a few commands which ignore any counts (such as ^R), the rest of the editor commands use a count to indicate a simple repetition of their effect. Thus 5w advances five words on the current line, while 5RETURN advances five lines. A very useful instance of a count as a repetition is a count given to the . command, which repeats the last changing command. If you do dw and then 3., you will delete first one and then three words. You can then delete two more words with 2..

8.3. More file manipulation commands

The following table lists the file manipulation commands which you can use when you are in *vi*. All of these commands are followed by a CR or ESC. The most basic commands are :w and :e. A normal editing session on a single file will end with a ZZ command. If you are editing for a long period of time you can give :w commands occasionally after major amounts of editing, and then finish with a ZZ. When you edit more than one file, you can finish with one

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

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

with a :w and start editing a new file by giving a :e command, or set *autowrite* and use :n <file>.

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

The :e command can be given a + argument to start at the end of the file, or a +*n* argument to start at line *n*. In actuality, *n* may be any editor command not containing a space, usefully a scan like +/*pat* or +?*pat*. In forming new names to the e command, you can use the character % which is replaced by the current file name, or the character # which is replaced by the alternate file name. The alternate file name is generally the last name you typed other than the current file. Thus if you try to do a :e and get a diagnostic that you haven't written the file, you can give a :w command and then a :e # command to redo the previous :e.

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

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

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

If you are editing large programs, you will find the :ta command very useful. It utilizes a data base of function names and their locations, which can be created by programs such as *ctags*, to quickly find a function whose name you give. If the :ta command will require the editor to switch files, then you must :w or abandon any changes before switching. You can repeat the :ta command without any arguments to look for the same tag again. (The tag feature is not available in some v2 editors.)

8.4. More about searching for strings

When you are searching for strings in the file with / and ?, the editor normally places you at the next or previous occurrence of the string. If you are using an operator such as d, c or y, then you may well wish to affect lines up to the line before the line containing the pattern.

You can give a search of the form */pat/−n* to refer to the *n*'th line before the next line containing *pat*, or you can use + instead of − to refer to the lines after the one containing *pat*. If you don't give a line offset, then the editor will affect characters up to the match place, rather than whole lines; thus use "+0" to affect to the line which matches.

You can have the editor ignore the case of words in the searches it does by giving the command `:se ic`. The command `:se noic` 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 *w* 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 :fCR, printing the current file, whether it has been modified, the current line number and the number of lines in the file, and the percentage of the way through the file that you are.

^H (BS) Same as left arrow. (See h). During an insert, eliminates the last input character, backing over it but not erasing it; it remains so you can see what you typed if you wish to type something only slightly different (3.1, 7.5).

^I (TAB) Not a command character. When inserted it prints as some number of spaces. When the cursor is at a tab character it rests at the last of the spaces which represent the tab. The spacing of tabstops is controlled by the *tabstop* option (4.1, 6.6).

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

^K Unused.

^L The ASCII formfeed character, this causes the screen to be cleared and redrawn. This is useful after a transmission error, if characters typed by a program other than the editor scramble the screen, or after output is stopped by an interrupt (5.4, 7.2f).

^M (CR) A carriage return advances to the next line, at the first non-white position in the line. Given a count, it advances that many lines (2.3). During an insert, a CR causes the insert to continue onto another line (3.1).

^N Same as down arrow (see j).

^O Unused.

^P — Same as up arrow (see k).

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

^R — Redraws the current screen, eliminating logical lines not corresponding to physical lines (lines with only a single @ character on them). On hardcopy terminals in *open* mode, retypes the current line (5.4, 7.2, 7.8).

^S — Unused. Some teletype drivers use ^S to suspend output until ^Qis

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

^U — Scrolls the screen up, inverting ^D which scrolls down. Counts work as they do for ^D, and the previous scroll amount is common to both. On a dumb terminal, ^U will often necessitate clearing and redrawing the screen further back in the file (2.1, 7.2).

^V — Not a command character. In input mode, quotes the next character so that it is possible to insert non-printing and special characters into the file (4.2, 7.5).

^W — Not a command character. During an insert, backs up as b would in command mode; the deleted characters remain on the display (see ^H) (7.5).

^X — Unused.

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

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

^[(ESC). — Cancels a partially formed command, such as a z when no following character has yet been given; terminates inputs on the last line (read by commands such as : / and ?); ends insertions of new text into the buffer. If an ESC is given when quiescent in command state, the editor rings the bell or flashes the screen. You can thus hit ESC if you don't know what is happening till the editor rings the bell. If you don't know if you are in insert mode you can type ESCa, and then material to be input; the material will be inserted correctly whether or not you were in insert mode when you started (1.5, 3.1, 7.5).

^\ — Unused.

^] — Searches for the word which is after the cursor as a tag. Equivalent to typing :ta, this word, and then a CR. Mnemonically, this command is "go right to" (7.3).

^^ — Equivalent to :e #CR, returning to the previous position in the last edited file, or editing a file which you specified if you got a 'No write since last change diagnostic' and do not want to have to type the file name again (7.3). (You have to do a :w before ^^ will work in this case. If you do not wish to write the file you should do :e! #CR instead.)

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

SPACE — Same as right arrow (see l).

! — An operator, which processes lines from the buffer with reformatting commands. Follow ! with the object to be processed, and then the command name terminated by CR. Doubling ! and preceding it by a count causes count lines to be filtered; otherwise the count is passed on to the object after the !. Thus 2!}*fmt*CR reformats the next two paragraphs by running them through the program *fmt*. If you are working on LISP, the command !%*grind*CR,* given at the

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

beginning of a function, will run the text of the function through the LISP grinder (6.7, 7.3). To read a file or the output of a command into the buffer use :r (7.3). To simply execute a command use :! (7.3).

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

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

$ Moves to the end of the current line. If you use :set list, 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 3cw and c3w 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 S 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)

Edit: A Tutorial

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 casual users.

This edition documents Version 2 of *edit* and *ex*.

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

Edit: A Tutorial

Ricki Blau

James Joyce

Computing Services
University of California
Berkeley, California 94720

Text editing using a terminal connected to a computer allows one to create, modify, and print text easily. A specialized computer program, known as a *text editor*, assists in creating and revising text. Creating text is very much like typing on an electric typewriter. Modifying text involves telling the text editor what to add, change, or delete. Text is printed by giving a command to print the file contents, with or without special instructions as to the format of the desired output.

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

program — A set of instructions given to the computer, describing the sequence of steps which the computer performs in order to accomplish a specific task. As an example, a series of steps to balance your checkbook is a program.

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

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

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

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

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

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

Session 1: Creating a File of Text

To use the editor you must first make contact with the computer by logging in to UNIX. We'll quickly review the standard UNIX login procedure.

If the terminal you are using is directly linked to the computer, turn it on and press carriage return, usually labelled "RETURN". If your terminal connects with the computer over a telephone line, turn on the terminal, dial the system access number, and, when you hear a high-pitched tone, place the receiver of the telephone in the acoustic coupler. Press carriage return once and await the login message:

login:

Type your login name, which identifies you to UNIX, on the same line as the login message, and press carriage return. If the terminal you are using has both upper and lower case, be sure you enter your login name in lower case; otherwise UNIX assumes your terminal has only upper case and will not recognize lower case letters you may type. UNIX types "login:" and you reply with your login name, for example "susan":

login: **susan** *(and press carriage return)*

(In the examples, input typed by the user appears in bold face to distinguish it from the responses from UNIX.)

UNIX will next respond with a request for a password as an additional precaution to prevent unauthorized people from using your account. The password will not appear when you type it to prevent others from seeing it. The message is:

Password: *(type your password and press carriage return)*

If any of the information you gave during the login sequence was mistyped or incorrect, UNIX will respond with

Login incorrect
login:

in which case you should start the login process anew. Assuming that you have successfully logged in, UNIX will print the message of the day and eventually will present you with a % at the beginning of a fresh line. The % is the UNIX prompt symbol which tells you that UNIX is ready to accept a command.

Asking for *edit*

You are ready to tell UNIX that you want to work with edit, the text editor. Now is a convenient time to choose a name for the file of text which you are about to create. To begin your editing session type edit followed by a space and then the filename which you have selected, for example "text". When you have completed the command, press carriage return and wait for edit's response:

```
% edit text    (followed by a carriage return)
"text" No such file or directory
```

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

```
"text" No such file or directory
```

On the next line appears edit's prompt ":", announcing that edit expects a command from you. You may now begin to create the new file.

The "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, so you may enter the correct command.

A summary

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

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

Entering text

You may now begin to enter text into the buffer. This is done by *appending* text to whatever is currently in the buffer. Since there is nothing in the buffer at the moment, you are appending text to nothing; in effect, you are creating text. Most edit commands have two forms: a word which describes what the command does and a shorter abbreviation of that word. Either form may be used. Many beginners find the full command names easier to remember, but once you are familiar with editing you may prefer to type the shorter abbreviations. The command to input text is "append" which may be abbreviated "a". Type append and press carriage return.

```
% edit text
:append
```

Messages from *edit*

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

"append" or "a", you will receive this message:

```
:add
add: Not an editor command
```

When you receive a diagnostic message, check what you typed in order to determine what part of your command confused edit. The message above means that edit was unable to recognize your mistyped command and, therefore, did not execute it. Instead, a new ":" appeared to let you know that edit is again ready to receive a command.

Text input mode

By giving the command "append" (or using the abbreviation "a"), you entered *text input mode*, also known as *append mode*. When you enter text input mode, edit responds by doing nothing. You will not receive any prompts while in text input mode. This is your signal that you are to begin entering lines of text. You can enter pretty much anything you want on the lines. The lines are transmitted one by one to the buffer and held there during the editing session. You may append as much text as you want, and *when you wish to stop entering text lines you should type a period as the only character on the line and press carriage return.* When you give this signal that you want to stop appending text, you will exit from text input mode and reenter command mode. Edit will again prompt you for a command by printing ":"

Leaving append mode does not destroy the text in the buffer. You have to leave append mode to do any of the other kinds of editing, such as changing, adding, or printing text. If you type a period as the first character and type any other character on the same line, edit will believe you want to remain in append mode and will not let you out. As this can be very frustrating, be sure to type only the period and carriage return.

This is as good a place as any to learn an important lesson about computers and text: a blank space is a character as far as a computer is concerned. If you so much as type a period followed by a blank (that is, type a period and then the space bar on the keyboard), you will remain in append mode with the last line of text being:

Let's say that the lines of text you enter are (try to type exactly what you see, including "thiss"):

```
This is some sample text.
And thiss is some more text.
Text editing is strange, but nice.
```

The last line is the period followed by a carriage return that gets you out of append mode. If while typing the line you hit an incorrect key, recall that you may delete the incorrect character or cancel the entire line of input by erasing in the usual way. Refer to "Communicating with UNIX" if you need to review the procedures for making a correction. Erasing a character or cancelling a line must be done before the line has been completed by a carriage return. We will discuss changes in lines already typed in session 2.

Writing text to disk

You are now ready to edit the text. The simplest kind of editing is to write it to disk as a file for safekeeping after the session is over. This is the only way to save information from one session to the next, since the editor's buffer is temporary and will last only until the end of the editing session. Thus, learning how to write a file to disk is second in importance only to entering the text. To write the contents of the buffer to a disk file, use the command "write" (or its abbreviation "w"):

```
:write
```

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

Edit must have a filename to use before it can write a file. If you forgot to indicate the name of the file when you began the editing session, edit will print

```
No current filename
```

in response to your write command. If this happens, you can specify the filename in a new write command:

```
:write text
```

After the "write" (or "w") type a space and then the name of the file.

Logging off

We have done enough for this first lesson on using the UNIX text editor, and are ready to quit the session with edit. To do this we type "quit" (or "q") and press carriage return:

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

The % is from UNIX to tell you that your session with edit is over and you may command UNIX further. Since we want to end the entire session at the terminal we also need to exit from UNIX. In response to the UNIX prompt of "%" type the command **logout** or a "control d" This is done by holding down the control key (usually labelled "CTRL") and simultaneously pressing the d key. This will end your session with UNIX and will ready the terminal for the next user. It is always important to 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)
%
```

This time when you say that you want to edit, you can specify the name of the file you worked on last time. This will start edit working and it will fetch the contents of the file into the buffer, so that you can resume editing the same file. When edit has copied the file into the buffer, it will repeat its name and tell you the number of lines and characters it contains. Thus,

```
% edit text
"text" 3 lines, 90 characters
```

means you asked edit to fetch the file named "text" for editing, causing it to copy the 90 characters of text into the buffer. Edit awaits your further instructions. In this session, we will append more text to our file, print the contents of the buffer, and learn to change the text of a line.

Adding more text to the file

If you want to add more to the end of your text you may do so by using the append command to enter text input mode. When append is the first command of your editing session, the lines you enter are placed at the end of the buffer. We'll soon discuss why this happens. Here we'll use the abbreviation for the append command, "a":

```
:a
This is text added in Session 2.
It doesn't mean much here, but
it does illustrate the editor.
```

Interrupt

Should you press the RUBOUT key (sometimes labelled DELETE) while working with edit, it will send this message to you:

```
Interrupt
```

Any command that edit might be executing is terminated by rubout or delete, causing edit to prompt you for a new command. If you are appending text at the time, you will exit from append mode and be expected to give another command. The line of text that you were typing when the append command was interrupted will not be entered into the buffer.

Making corrections

If you have read a general introduction to UNIX, such as "Communicating with UNIX" you will recall that it is possible to erase individual letters that you have typed. This is done by typing the designated erase character as many times as there are characters you want to erase. Accounts normally start out using the number sign (#) as the erase character, but it's possible for a different erase character to be selected†. We'll show "#" as the erase character in our

† UNIX accounts may be "personalized" in other ways, too. If you're using an established account, check with someone who is familiar with your account to find out if it has any other non-standard characteristics which may affect your work. Accounts for students in classes are often given class commands and other special features; the teaching assistant or instructor is the best source of information about these changes.

examples, but if you've changed your erase character to backspace (control-H) or something else, be sure to use your own erase character.

If you make a bad start in a line and would like to begin again, erasing individual characters with a "#" is cumbersome — what if you had 15 characters in your line and wanted to get rid of them? To do so either requires:

```
This is yukky tex################
```

with no room for the great text you'd like to type, or,

```
This is yukky tex@This is great text.
```

When you type the at-sign (@), you erase the entire line typed so far. (An account may select a different line erase character to use in place of @. If your line erase character has been changed, use it where the examples show "@".) You may immediately begin to retype the line. This, unfortunately, does not help after you type the line and press carriage return. To make corrections in lines which have been completed, it is necessary to use the editing commands covered in this session and those that follow.

Listing what's in the buffer

Having appended text to what you wrote in Lesson 1, you might be curious to see what is in the buffer. To print the contents of the buffer, type the command:

```
:1,$p
```

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

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

Occasionally, you may enter into the buffer a character which can't be printed, which is done by striking a key while the CTRL key is depressed. In printing lines, edit uses a special notation to show the existence of non-printing characters. Suppose you had introduced the non-printing character "control-A" into the word "illustrate" by accidently holding down the CTRL key while typing "a". Edit would display

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

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

In looking over the text we see that "this" is typed as "thiss" in the second line, as suggested. Let's correct the spelling.

Finding things in the buffer

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

```
:/thiss/
```

By typing /thiss/ and pressing carriage return edit is instructed to search for "thiss". If we asked edit to look for a pattern of characters which it could not find in the buffer, it would respond "Pattern not found" When edit finds the characters "thiss", it will print the line of text for your inspection:

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

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

The current line

At all times during an editing session, edit keeps track of the line in the buffer where it is positioned. In general, the line which has been most recently printed, entered, or changed is considered to be the current position in the buffer. The editor is prepared to make changes at the current position in the buffer, unless you direct it to act in another location. When you bring a file into the editor, you will be positioned at the last line in the file. If your initial editing command is "append", the lines you enter are added to the end of the file, that is, they are placed after the current position. You can refer to your current position in the buffer by the symbol period (.) usually known by the name "dot". If you type ". " and carriage return you will be instructing edit to print the current line:

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

If you want to know the number of the current line, you can type .= and carriage return, and edit will respond with the line number:

```
:.=
2
```

If you type the number of any line and a carriage return, edit will position you at that line and print its contents:

```
:2
And thiss is some more text.
```

You should experiment with these commands to assure yourself that you understand what they do.

Numbering lines (nu)

The number (nu) command is similar to print, giving both the number and the text of each printed line. To see the number and the text of the current line type

```
:nu
     2  And thiss is some more text.
```

Notice that the shortest abbreviation for the number command is "nu" (and not "n" which is used for a different command). You may specify a range of lines to be listed by the number command in the same way that lines are specified for print. For example, "1,$nu" lists all lines in the buffer with the corresponding line numbers.

Substitute command (s)

Now that we have found our misspelled word it is time to change it from "thiss" to "this" As far as edit is concerned, changing things is a matter of substituting one thing for another. As *a* stood for *append*, so *s* stands for *substitute*. We will use the abbreviation "s" to reduce the chance of mistyping the substitute command. This command will instruct edit to make the change:

2s/thiss/this/

We first indicate the line to be changed, line 2, and then type an "s" to indicate we want substitution. Inside the first set of slashes are the characters that we want to change, followed by the characters to replace them and then a closing slash mark. To summarize:

2s/ *what is to be changed / what to change to /*

If edit finds an exact match of the characters to be changed it will make the change only in the first occurrence of the characters. If it does not find the characters to be changed it will respond:

Substitute pattern match failed

indicating your instructions could not be carried out. When edit does find the characters which you want to change, it will make the substitution and automatically print the changed line, so that you can check that the correct 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 positioned ("."). In this case, the command without a line number would have produced the same result because we were already positioned at the line we wished to change.

For another illustration of substitution we may choose the line:

Text editing is strange, but nice.

We might like to be a bit more positive. Thus, we could take out the characters "strange, but " so the line would read:

Text editing is nice.

A command which will first position edit at that line and then make the substitution is:

:/strange/s/strange, but //

What we have done here is combine our search with our substitution. Such combinations are perfectly legal. This illustrates that we do not necessarily have to use line numbers to identify a line to edit. Instead, we may identify the line we want to change by asking edit to search for a specified pattern of letters which occurs in that line. The parts of the above command are:

/strange/	tells edit to find the characters "strange" in the text
s	tells edit we want to make a substitution
/strange, but //	substitutes nothing at all for the characters "strange, but "

You should note the space after "but" in "/strange, but /". If you do not indicate the space is to be taken out, your line will be:

Text editing is nice.

which looks a little funny because of the extra space between "is" and "nice" Again, we realize from this that a blank space is a real character to a computer, and in editing text we need to be aware of spaces within a line just as we would be aware of an "a" or a "z"

Another way to list what's in the buffer (z)

Although the print command is useful for looking at specific lines in the buffer, other commands can be more convenient for viewing large sections of text. You can ask to see a screen full of text at a time by using the command z. If you type

```
:1z
```

edit will start with line 1 and continue printing lines, stopping either when the screen of your terminal is full or when the last line in the buffer has been printed. If you want to read the next segment of text, give the command

```
:z
```

If no starting line number is given for the z command, printing will start at the "current" line, in this case the last line printed. Viewing lines in the buffer one screen full at a time is known as paging. Paging can also be used to print a section of text on a hard-copy terminal.

Saving the modified text

This seems to be a good place to pause in our work, and so we should end the second session. If you (in haste) type "q" to quit the session your dialogue with edit will be:

```
:q
No write since last change (q! quits)
```

This is edit's warning that you have not written the modified contents of the buffer to disk. You run the risk of losing the work you have done during the editing session since the latest write command. Since in this lesson we have not written to disk at all, everything we have done would be lost. If we did not want to save the work done during this editing session, we would have to type "q!" to confirm that we indeed wanted to end the session immediately, losing the contents of the buffer. However, since we want to preserve what we have edited, we need to say:

```
:w
"text" 6 lines, 171 characters
```

and then,

```
:q
% logout
```

and hang up the phone or turn off the terminal when UNIX asks for a login name. This is the end of the second session on UNIX text editing.

Session 3

Bringing text into the buffer (e)

Login to UNIX and make contact with edit. You should try to login without looking at the notes, but if you must then by all means do.

Did you remember to give the name of the file you wanted to edit? That is, did you say

```
% edit text
```

or simply

```
% edit
```

Both ways get you in contact with edit, but the first way will bring a copy of the file named "text" into the buffer. If you did forget to tell edit the name of your file, you can get it into the buffer by saying:

```
:e text
"text" 6 lines, 171 characters
```

The command edit, which may be abbreviated "e" when you're in the editor, tells edit that you want to erase anything that might already be in the buffer and bring a copy of the file "text" into the buffer for editing. You may also use the edit (e) command to change files in the middle of an editing session or to give edit the name of a new file that you want to create. Because the edit command clears the buffer, you will receive a warning if you try to edit a new file without having saved a copy of the old file. This gives you a chance to write the contents of the buffer to disk before editing the next file.

Moving text in the buffer (m)

Edit allows you to move lines of text from one location in the buffer to another by means of the move (m) command:

```
:2,4m$
```

This command directs edit to move lines 2, 3, and 4 to the end of the buffer ($). The format for the move command is: that you specify the first line to be moved, the last line to be moved, the move command "m", and the line after which the moved text is to be placed. Thus,

```
:1,6m20
```

would instruct edit to move lines 1 through 6 (inclusive) to a position after line 20 in the buffer. To move only one line, say, line 4, to a position in the buffer after line 6, the command would be "4m6".

Let's move some text using the command:

```
:5,$m1
2 lines moved
it does illustrate the editor.
```

After executing a command which changes more than one line of the buffer, edit tells how many lines were affected by the change. The last moved line is printed for your inspection. If you want to see more than just the last line, use the print (p), z, or number (nu) command to view more text. The buffer should now contain:

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

We can restore the original order by typing:

```
:4,5m1
```

or, combining context searching and the move command:

```
:/And this is some/,./This is text/m/This is some sample/
```

The problem with combining context searching with the move command is that the chance of making a typing error in such a long command is greater than if one types line numbers.

Copying lines (copy)

The copy command is used to make a second copy of specified lines, leaving the original lines where they were. Copy has the same format as the move command, for example:

```
:12,15copy $
```

makes a copy of lines 12 through 15, placing the added lines after the buffer's end ($). Experiment with the copy command so that you can become familiar with how it works. Note that the shortest abbreviation for copy is "co" (and not the letter "c" which has another meaning).

Deleting lines (d)

Suppose you want to delete the line

```
This is text added in Session 2.
```

from the buffer. If you know the number of the line to be deleted, you can type that number followed by "delete" or "d". This example deletes line 4:

```
:4d
It doesn't mean much here, but
```

Here "4" is the number of the line to be deleted and "delete" or "d" is the command to delete the line. After executing the delete command, edit prints the line which has become the current line (".").

If you do not happen to know the line number you can search for the line and then delete it using this sequence of commands:

```
:/added in Session 2./
This is text added in Session 2.
:d
It doesn't mean much here, but
```

The "/added in Session 2./" asks edit to locate and print the next line which contains the indicated text. Once you are sure that you have correctly specified the line that you want to delete, you can enter the delete (d) command. In this case it is not necessary to specify a line number before the "d". If no line number is given, edit deletes the current line ("."), that is, the line found by our search. After the deletion, your buffer should contain:

```
This is some sample text.
And this is some more text.
Text editing is nice.
It doesn't mean much here, but
it does illustrate the editor.
```

To delete both lines 2 and 3:

```
And this is some more text.
Text editing is nice.
```

you type

```
:2,3d
```

which specifies the range of lines from 2 to 3, and the operation on those lines — "d" for delete.

Again, this presumes that you know the line numbers for the lines to be deleted. If you do not you might combine the search command with the delete command as so:

```
:/And this is some/,/Text editing is nice./d
```

This tells the editor to start deleting with the next line that contains the characters "And this is some" and continue until it has deleted the line containing "Text editing is nice."

A word or two of caution:

In using the search function to locate lines to be deleted you should be absolutely sure the characters you give as the basis for the search will take edit to the line you want deleted. Edit will search for the first occurrence of the characters starting from where you last edited — that is, from the line you see printed if you type dot (.).

A search based on too few characters may result in the wrong lines being deleted, which edit will do as easily as if you had meant it. For this reason, it is usually safer to specify the search and then delete in two separate steps, at least until you become familiar enough with using the editor that you understand how best to specify searches. For a beginner it is not a bad idea to double-check each command before pressing carriage return to send the command on its way.

Undo (u) to the rescue

The undo (u) command has the ability to reverse the effects of the last command. To undo the previous command, type "u" or "undo". Undo can rescue the contents of the buffer from many an unfortunate mistake. However, its powers are not unlimited, so it is still wise to be reasonably careful about the commands you give. It is possible to undo only commands which have the power to change the buffer, for example delete, append, move, copy, substitute, and even undo itself. The commands write (w) and edit (e) which interact with disk files cannot be undone, nor can commands such as print which do not change the buffer. Most importantly, the only command which can be reversed by undo is the last "undo-able" command which you gave.

To illustrate, let's issue an undo command. Recall that the last buffer-changing command we gave deleted the lines which were formerly numbered 2 and 3. Executing undo at this moment will reverse the effects of the deletion, causing those two lines to be replaced in the buffer.

```
:u
2 more lines in file after undo
And this is some more text.
```

Here again, edit informs you if the command affects more than one line, and prints the text of the line which is now "dot" (the current line).

More about the dot (.) and buffer end (S)

The function assumed by the symbol dot depends on its context. It can be used:

1. to exit from append mode we type dot (and only a dot) on a line and press carriage return;
2. to refer to the line we are at in the buffer.

Dot can also be combined with the equal sign to get the number of the line currently being edited:

```
:.=
```

Thus if we type ".=" we are asking for the number of the line and if we type "." we are asking for the text of the line.

In this editing session and the last, we used the dollar sign to indicate the end of the buffer in commands such as print, copy, and move. The dollar sign as a command asks edit to print the last line in the buffer. If the dollar sign is combined with the equal sign (S=) edit will print the line number corresponding to the last line in the buffer.

"." and "S" therefore represent line numbers. Whenever appropriate, these symbols can be used in place of line numbers in commands. For example

```
:.,Sd
```

instructs edit to delete all lines from the current line (.) to the end of the buffer.

Moving around in the buffer (+ and −)

It is frequently convenient during an editing session to go back and re-read a previous line. We could specify a context search for a line we want to read if we remember some of its text, but if we simply want to see what was written a few, say 3, lines ago, we can type

```
-3p
```

This tells edit to move back to a position 3 lines before the current line (.) and print that line. We can move forward in the buffer similarly:

```
+2p
```

instructs edit to print the line which is 2 ahead of our current position.

You may use "+" and "−" in any command where edit accepts line numbers. Line numbers specified with "+" or "−" can be combined to print a range of lines. The command

```
:-1,+2copyS
```

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 (S).

Try typing only "−"; you will move back one line just as if you had typed "−1p" Typing the command "+" works similarly. You might also try typing a few plus or minus signs in a row (such as "+++") to see edit's response. Typing a carriage return alone on a line is the equivalent of typing "+1p"; it will move you one line ahead in the buffer and print that line.

If you are at the last line of the buffer and try to move further ahead, perhaps by typing a "+" or a carriage return alone on the line, edit will remind you that you are at the end of the buffer:

```
At end-of-file
```

Similarly, if you try to move to a position before the first line, edit will print one of these messages:

Nonzero address required on this command
Negative address — first buffer line is 1

The number associated with a buffer line is the line's "address", in that it can be used to locate the line.

Changing lines (c)

There may be occasions when you want to delete certain lines and insert new text in their place. This can be accomplished easily with the change (c) command. The change command instructs edit to delete specified lines and then switch to text input mode in order to accept the text which will replace them. Let's say we want to change the first two lines in the buffer:

This is some sample text.
And this is some more text.

to read

This text was created with the UNIX text editor.

To do so, you can type:

```
:1,2c
2 lines changed
This text was created with the UNIX text editor.
```

In the command 1,2c we specify that we want to change the range of lines beginning with 1 and ending with 2 by giving line numbers as with the print command. These lines will be deleted. After a carriage return enters the change command, edit notifies you if more than one line will be changed and places you in text input mode. Any text typed on the following lines will be inserted into the position where lines were deleted by the change command. You will remain in text input mode until you exit in the usual way, by typing a period alone on a line. Note that the number of lines added to the buffer need not be the same as the number of lines deleted.

This is the end of the third session on text editing with UNIX.

Session 4

This lesson covers several topics, starting with commands which apply throughout the buffer, characters with special meanings, and how to issue UNIX commands while in the editor. The next topics deal with files: more on reading and writing, and methods of recovering files lost in a crash. The final section suggests sources of further information.

Making commands global (g)

One disadvantage to the commands we have used for searching or substituting is that if you have a number of instances of a word to change it appears that you have to type the command repeatedly, once for each time the change needs to be made. Edit, however, provides a way to make commands apply to the entire contents of the buffer — the **global** (g) command.

To print all lines containing a certain sequence of characters (say, "text") the command is:

```
:g/text/p
```

The "g" instructs edit to make a global search for all lines in the buffer containing the characters "text". The "p" prints the lines found.

To issue a global command, start by typing a "g" and then a search pattern identifying the lines to be affected. Then, on the same line, type the command to be executed on the identified lines. Global substitutions are frequently useful. For example, to change all instances of the word "text" to the word "material" the command would be a combination of the global search and the substitute command:

```
:g/text/s/text/material/g
```

Note the "g" at the end of the global command which instructs edit to change each and every instance of "text" to "material". If you do not type the "g" at the end of the command only the *first* instance of "text" in each line will be changed (the normal result of the substitute command). The "g" at the end of the command is independent of the "g" at the beginning. You may give a command such as:

```
:14s/text/material/g
```

to change every instance of "text" in line 14 alone. Further, neither command will change "Text" to "material" because "Text" begins with a capital rather than a lower-case *t*.

Edit does not automatically print the lines modified by a global command. If you want the lines to be printed, type a "p" at the end of the global command:

```
:g/text/s/text/material/gp
```

The usual qualification should be made about using the global command in combination with any other — in essence, be sure of what you are telling edit to do to the entire buffer. For example,

```
:g/ /d
72 less lines in file after global
```

will delete every line containing a blank anywhere in it. This could adversely affect your document, since most lines have spaces between words and thus would be deleted. After executing the global command, edit will print a warning if the command added or deleted more than one line. Fortunately, the undo command can reverse the effects of a global command. You should experiment with the global command on a small buffer of text to see what it can do for you.

More about searching and substituting

In using slashes to identify a character string that we want to search for or change, we have always specified the exact characters. There is a less tedious way to repeat the same string of characters. To change "noun" to "nouns" we may type either

```
:/noun/s/noun/nouns/
```

as we have done in the past, or a somewhat abbreviated command:

```
:/noun/s//nouns/
```

In this example, the characters to be changed are not specified — there are no characters, not even a space, between the two slash marks which indicate what is to be changed. This lack of characters between the slashes is taken by the editor to mean "use the characters we last searched for as the characters to be changed."

Similarly, the last context search may be repeated by typing a pair of slashes with nothing between them:

```
:/does/
It doesn't mean much here, but
://
it does illustrate the editor.
```

Because no characters are specified for the second search, the editor scans the buffer for the next occurrence of the characters "does".

Edit normally searches forward through the buffer, wrapping around from the end of the buffer to the beginning, until the specified character string is found. If you want to search in the reverse direction, use question marks (?) instead of slashes to surround the character string.

It's 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/noun/nouns/
```

we could use the command

```
:/nouns/&
```

or simply

```
://&
```

to make the same change on the next line in the buffer containing the characters "nouns"

Special characters

Two characters have special meanings when used in specifying searches: "$" and "^". "$" is taken by the editor to mean "end of the line" and is used to identify strings which occur at the end of a line.

```
:g/ing$/s//ed/p
```

tells the editor to search for all lines ending in "ing" (and nothing else, not even a blank space), to change each final "ing" to "ed" and print the changed lines.

The symbol "^" indicates the beginning of a line. Thus,

```
:s/^/1. /
```

instructs the editor to insert "1." and a space at the beginning of the current line.

The characters "$" and "^" have special meanings only in the context of searching. At

other times, they are ordinary characters. If you ever need to search for a character that has a special meaning, you must indicate that the character is to temporarily lose its special significance by typing another special character, the backslash (\), before it.

```
:s/\$/dollar/
```

looks for the character "$" in the current line and replaces it by the word "dollar". Were it not for the backslash, the "$" would have represented "the end of the line" in your search, rather than the character "$". The backslash retains its special significance unless it is preceded by another backslash.

Issuing UNIX commands from the editor

After creating several files with the editor, you may want to delete files no longer useful to you or ask for a list of your files. Removing and listing files are not functions of the editor, and so they require the use of UNIX system commands (also referred to as "shell" commands, as "shell" is the name of the program that processes UNIX commands). You do not need to quit the editor to execute a UNIX command as long as you indicate that it is to be sent to the shell for execution. To use the UNIX command *rm* to remove the file named "junk" type:

```
:!rm junk
!
```

The exclamation mark (!) indicates that the rest of the line is to be processed as a UNIX command. If the buffer contents have not been written since the last change, a warning will be printed before the command is executed. The editor prints a "!" when the command is completed. The tutorial "Communicating with UNIX" describes useful features of the system, of which the editor is only one part.

Filenames and file manipulation

Throughout each editing session, edit keeps track of the name of the file being edited as the *current filename*. Edit remembers as the current filename the name given when you entered the editor. The current filename changes whenever the edit (e) command is used to specify a new file. Once edit has recorded a current filename, it inserts that name into any command where a filename has been omitted. If a write command does not specify a file, edit, as we have seen, supplies the current filename. You can have the editor write onto a different file by including its name in the write command:

```
:w chapter3
"chapter3" 283 lines, 8698 characters
```

The current filename remembered by the editor *will not be changed as a result of the write command unless it is the first filename given in the editing session.* Thus, in the next write command which does not specify a name, edit will write onto the current file and not onto the file "chapter3".

The file (f) command

To ask for the current filename, type file (or f). In response, the editor provides current information about the buffer, including the filename, your current position, and the number of lines in the buffer:

```
:f
"text" [Modified] line 3 of 4 --75%--
```

If the contents of the buffer have changed since the last time the file was written, the editor will tell you that the file has been "[Modified]". After you save the changes by writing onto a disk file, the buffer will no longer be considered modified:

```
:w
"text" 4 lines, 88 characters
:f
"text" line 3 of 4 --75%--
```

Reading additional files (r)

The read (r) command allows you to add the contents of a file to the buffer without destroying the text already there. To use it, specify the line after which the new text will be placed, the command *r*, and then the name of the file.

```
:$r bibliography
"bibliography" 18 lines, 473 characters
```

This command reads in the file *bibliography* and adds it to the buffer after the last line. The current filename is not changed by the read command unless it is the first filename given in the editing session.

Writing parts of the buffer

The write (w) command can write all or part of the buffer to a file you specify. We are already familiar with writing the entire contents of the buffer to a disk file. To write only part of the buffer onto a file, indicate the beginning and ending lines before the write command, for example

```
:45,$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.

Recovering files

Under most circumstances, edit's crash recovery mechanism is able to save work to within a few lines of changes after a crash or if the phone is hung up accidently. If you lose the contents of an editing buffer in a system crash, you will normally receive mail when you login which gives the name of the recovered file. To recover the file, enter the editor and type the command recover (rec), followed by the name of the lost file.

```
:recover 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.

Other recovery techniques

If something goes wrong when you are using the editor, it may be possible to save your work by using the command preserve (pre), which saves the buffer as if the system had crashed. If you are writing a file and you get the message "Quota exceeded", you have tried to use more disk storage than is allotted to your account. *Proceed with caution* because it is likely that only a part of the editor's buffer is now present in the file you tried to write. In this case you should use the shell escape from the editor (!) to remove some files you don't need and try to write the file again. If this is not possible and you cannot find someone to help you, enter the command

```
:preserve
```

and then seek help. Do not simply leave the editor. If you do, the buffer will be lost, and you may not be able to save your file. After a preserve, you can use the recover command once the problem has been corrected.

If you make an undesirable change to the buffer and issue a write command before discovering your mistake, the modified version will replace any previous version of the file. Should you ever lose a good version of a document in this way, do not panic and leave the editor. As long as you stay in the editor, the contents of the buffer remain accessible. Depending on the nature of the problem, it may be possible to restore the buffer to a more complete state with the undo command. After fixing the damaged buffer, you can again write the file to disk.

Further reading and other information

Edit is an editor designed for beginning and casual users. It is actually a version of a more powerful editor called ex. These lessons are intended to introduce you to the editor and its more commonly-used commands. We have not covered all of the editor's commands, just a selection of commands which should be sufficient to accomplish most of your editing tasks. You can find out more about the editor in the *Ex Reference Manual*, which is applicable to both *ex* and *edit*. The manual is available from the Computer Center Library, 213 Evans Hall. One way to become familiar with the manual is to begin by reading the description of commands that you already know.

Using *ex*

As you become more experienced with using the editor, you may still find that edit continues to meet your needs. However, should you become interested in using ex, it is easy to switch. To begin an editing session with ex, use the name ex in your command instead of edit.

Edit commands work the same way in ex, but the editing environment is somewhat different. You should be aware of a few differences that exist between the two versions of the editor. In edit, only the characters "^", "$", and "\" have special meanings in searching the buffer or indicating characters to be changed by a substitute command. Several additional characters have special meanings in ex, as described in the *Ex Reference Manual*. Another feature of the edit environment prevents users from accidently entering two alternative modes of editing, *open* and *visual*, in which the editor behaves quite differently than in normal command mode. If you are using ex and the editor behaves strangely, you may have accidently entered open mode by typing "o". Type the ESC key and then a "Q" to get out of open or visual mode and back into the regular editor command mode. The document *An Introduction to Display Editing with Vi* provides a full discussion of visual mode.

This tutorial was produced at the Computer Center of the University of California, Berkeley. We welcome comments and suggestions concerning this item and the UNIX documentation in general.

Index

Ex Reference Manual

Ex a line oriented text editor, which supports both command and display oriented editing. This reference manual describes the command oriented part of *ex*; the display editing features of *ex* are described in *An Introduction to Display Editing with Vi*. Other documents about the editor include the introduction *Edit: A Tutorial*, the *Ex/edit Command Summary*, and a *Vi Quick Reference* card.

Ex Reference Manual
Version 3.5/2.13 — September, 1980

William Joy

Revised for versions 1.5/2.13 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. If there is a TERMCAP variable in the environment, and the type of the terminal described there matches the TERM variable, then that description is used. Also if the TERMCAP variable contains a pathname (beginning with a /) then the editor will seek the description of the terminal in that file (rather than the default /etc/termcap.) If there is a variable EXINIT in the environment, then the editor will execute the commands in that variable, otherwise if there is a file *.exrc* in your HOME directory *ex* reads commands from that file, simulating a *source* command. Option setting commands placed in EXINIT or *.exrc* will be executed before each editor session.

A command to enter *ex* has the following prototype:†

ex [**−**] [**−v**] [**−t** *tag*] [**−r**] [**−l**] [**−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 *:* 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 :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 : 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.

Ex changes — Version 3.1 to 3.5

This update describes the new features and changes which have been made in converting from version 3.1 to 3.5 of *ex*. Each change is marked with the first version where it appeared.

Update to Ex Reference Manual

Command line options

3.4 A new command called *view* has been created. *View* is just like *vi* but it sets *readonly*.

3.4 The encryption code from the v7 editor is now part of *ex*. You can invoke *ex* with the −x option and it will ask for a key, as *ed*. The *ed x* command (to enter encryption mode from within the editor) is not available. This feature may not be available in all instances of *ex* due to memory limitations.

Commands

3.4 Provisions to handle the new process stopping features of the Berkeley TTY driver have been added. A new command, *stop*, takes you out of the editor cleanly and efficiently, returning you to the shell. Resuming the editor puts you back in command or visual mode, as appropriate. If *autowrite* is set and there are outstanding changes, a write is done first unless you say "stop!".

3.4 A

:vi <file>

command from visual mode is now treated the same as a

:edit <file> or :ex <file>

command. The meaning of the *vi* command from *ex* command mode is not affected.

3.3 A new command mode command *xit* (abbreviated *x*) has been added. This is the same as *wq* but will not bother to write if there have been no changes to the file.

Options

3.4 A read only mode now lets you guarantee you won't clobber your file by accident. You can set the on/off option *readonly* (*ro*), and writes will fail unless you use an ! after the write. Commands such as *x*, *ZZ*, the autowrite option, and in general anything that writes is affected. This option is turned on if you invoke ex with the −R flag.

3.4 The *wrapmargin* option is now usable. The way it works has been completely revamped. Now if you go past the margin (even in the middle of a word) the entire word is erased and rewritten on the next line. This changes the semantics of the number given to wrapmargin. 0 still means off. Any other number is still a distance from the right edge of the screen, but this location is now the right edge of the area where wraps can take place, instead of the left edge. Wrapmargin now behaves much like fill/nojustify mode in *nroff*.

3.3 The options *w300*, *w1200*, and *w9600* can be set. They are synonyms for *window*, but only apply at 300, 1200, or 9600 baud, respectively. Thus you can specify you want a 12 line window at 300 baud and a 23 line window at 1200 baud in your EXINIT with

:set w300=12 w1200=23

3.3 The new option *timeout* (default on) causes macros to time out after one second. Turn it off and they will wait forever. This is useful if you want multi character macros, but if your terminal sends escape sequences for arrow keys, it will be necessary to hit escape twice to get a beep.

3.3 The new option *remap* (default on) causes the editor to attempt to map the result of a macro mapping again until the mapping fails. This makes it possible, say, to map q to # and #1 to something else and get q1 mapped to something else. Turning it off makes it possible to map ^L to l and map ^R to ^L without having ^R map to L.

3.3 The new (string) valued option *tags* allows you to specify a list of tag files, similar to the "path" variable of csh. The files are separated by spaces (which are entered preceded by a backslash) and are searched left to right. The default value is "tags /usr/lib/tags", which has the same effect as before. It is recommended that "tags" always be the first entry. On Ernie CoVax, /usr/lib/tags contains entries for the system defined library procedures from section 3 of the manual.

Environment enquiries

3.4 The editor now adopts the convention that a null string in the environment is the same as not being set. This applies to TERM, TERMCAP, and EXINIT.

Vi Tutorial Update

Deleted features

3.3 The "q" command from visual no longer works at all. You must use "Q" to get to ex command mode. The "q" command was deleted because of user complaints about hitting it by accident too often.

3.5 The provisions for changing the window size with a numeric prefix argument to certain visual commands have been deleted. The correct way to change the window size is to use the z command, for example z5<cr> to change the window to 5 lines.

3.3 The option "mapinput" is dead. It has been replaced by a much more powerful mechanism: ":map!".

Change in default option settings

3.3 The default window sizes have been changed. At 300 baud the window is now 8 lines (it was 1/2 the screen size). At 1200 baud the window is now 16 lines (it was 2/3 the screen size, which was usually also 16 for a typical 24 line CRT). At 9600 baud the window is still the full screen size. Any baud rate less than 1200 behaves like 300, any over 1200 like 9600. This change makes *vi* more usable on a large screen at slow speeds.

Vi commands

3.3 The command "ZZ" from vi is the same as ":x<cr>". This is the recommended way to leave the editor. Z must be typed twice to avoid hitting it accidently.

3.4 The command ^Z is the same as ":stop<cr>". Note that if you have an arrow key that sends ^Z the stop function will take priority over the arrow function. If you have your "susp" character set to something besides ^Z, that key will be honored as well.

3.3 It is now possible from visual to string several search expressions together separated by semicolons the same as command mode. For example, you can say

```
/foo/;/bar
```

from visual and it will move to the first "bar" after the next "foo" This also works within one line.

3.3 ^R is now the same as ^L on terminals where the right arrow key sends ^L (This includes the Televideo 912/920 and the ADM 31 terminals.)

3.4 The visual page motion commands ^F and ^B now treat any preceding counts as number of pages to move, instead of changes to the window size. That is, 2^F moves forward 2 pages.

Macros

3.3 The "mapinput" mechanism of version 3.1 has been replaced by a more powerful mechanism. An "!" can follow the word "**map**" in the *map* command. **Map!**'ed macros only apply during input mode, while **map**'ed macros only apply during command mode. Using "**map**" or "**map!**" by itself produces a listing of macros in the corresponding mode.

3.4 A word abbreviation mode is now available. You can define abbreviations with the *abbreviate* command

:abbr foo find outer otter

which maps "foo" to "find outer otter". Abbreviations can be turned off with the *unabbreviate* command. The syntax of these commands is identical to the *map* and *unmap* commands, except that the ! forms do not exist. Abbreviations are considered when in visual input mode only, and only affect whole words typed in, using the conservative definition. (Thus "foobar" will not be mapped as it would using "map!") Abbreviate and unabbreviate can be abbreviated to "ab" and "una", respectively.

MED
MUNIX-Editor

Documentation-No.: D930058e

Best.-Nr.: D930058e
Author's initials: DK

Trademarks:

MUNIX,	for PCS
DEC, PDP	for DEC
UNIX	for Bell Laboratories

NAME

med – screen editor

SYNOPSIS

med file [*startline*] [*searchkey*]
med -
med

DESCRIPTION

Med calls the MED editor, which allows you to edit a file using the screen of your terminal and the cursor keys, somewhat like paper, pencil and eraser.

If you call the editor by *med file*, the first lines of the file will be shown on the terminal screen. If the *file* does not exist, it will be created.

Med called with no arguments continues where the previous *med* session ended.

Med called with argument "-" restores only one file from the last session.

The purpose of a screen editor is to create a new file, or to look at and change existing files. *Med* allows you to look through a file by moving a "window" (the text shown at the screen) over the files contents. The window may be moved up, down, to the left or the right. The "cursor" (a blinking underline or box) shows where *med* is at the moment. Initially *med* is in "replace mode", i.e. the characters typed will replace those under the cursor. The cursor may be moved around with the arrow keys [↑] [↓] [←] [→] or with [return], [home] or [tab].

To change text on the screen, the cursor has to be placed there and the new characters entered. [delch] deletes the character under the cursor. To insert characters in the middle of a word (like a 'd' in 'midle'), press [insert]; then new characters will be inserted, and the rest of the line will move to the right. Pressing [insert] again, *med* is switched back to "replace mode".

"Function keys" are special keys along the top or the right of the keyboard. Each function key does one job, such as moving some lines, looking for a word, or using another file. A picture of all the different function keys for your terminal should be attached to this description.

Some functions take arguments such as a number, a search string, a file name, etc. To enter an argument, type the sequence:

[enter] number or string [function].

[enter] leaves an '@' marker on the screen to remind you where the cursor was. It then goes to the bottom line on the screen to read the argument. This bottom line is also used to display error messages.

Function keys and their meaning

Move around:

↑ ↓ → ←	Move cursor
±tab	Skip to next/previous tab stop
±page	Move to next/previous page
±line	Move a halfpage up/down
±search	Search for a word
goto	Goto line *n*
left	Move window left
right	Move window right

Use another file or window:

use	Use another file
window	Create/delete a window
chwin	Go to next window

Cut and paste:

backspace	Delete character left of cursor
close	Delete line(s) into CLOSE buffer
delch	Delete character
open	Insert blank line(s)
pick	Copy text to PICK buffer
put	Get text from PICK buffer
restore	Get text from CLOSE buffer

Others:

do	Run a Unix command
enter	Enter a parameter
exit	Go back to Unix
insert	Flip insert mode
quote	Control character escape
refresh	Refresh
save	Write changes to disk
chtabs	Change tabs
macro	collect keystrokes

[enter] n [↑↓←→] Move the cursor **n** lines/columns in the direction of the arrow.

backspace	Remove the last character entered; thus typing "E R X backspace R" is the same as "E R R". This also works in insert mode.
chtab	Set a tab stop at the current position.
enter chtab	Remove the tab stop at the current position.
chwin	Go to the next window.
enter n chwin	Go to window **n**. Windows are numbered in the order they are created.
enter ↑↓←→ chwin	Argument defined by the cursor is used as window number.
close	Delete one line. It is saved in the CLOSE buffer.
enter close	Delete the rest of the line, replacing it with the line below.
enter n close	Delete **n** lines and save them in the CLOSE buffer.
enter ↑↓←→ close	Delete lines or a rectangle defined by the cursor and put it into the CLOSE buffer.
delch	Delete the character at the current cursor position, moving the following characters to the left.
do	Run the previous DO command exactly as it was given.
enter do	Take the current line as a command for the shell. Results are inserted below the current line.
enter cmd do	**cmd** is a Unix command in the format **[n[l]] prg [arg...]** (in shell notation). It replaces **n** paragraphs (or **n** lines if **l** appears) by the result of running filter **prg** on that text with given **args**. The replaced paragraphs are saved in the CLOSE buffer.
exit	Exit **med**, writing back the changed files to disk (a backup file named **.bak* is created).
enter a exit	Exit, but do not save files.
enter ad exit	Terminate with core dump.
goto	Move window to top of file.
enter goto	Move window to end of file.
enter n goto	Move to the **n**th line.
enter ↑↓←→ goto	Argument defined by the cursor is used as line number.
insert	Put the editor into insert mode. Subsequent characters are inserted at the cursor, i.e., characters to the right of the cursor are not replaced but moved to the right. Pressing insert again will place **med** back to replace mode.

left	Move window left (about a 1/3 window width).
enter left	Make the current column the last one (if possible).
enter n left	Move window n columns left.
+line	Move window down (about 1/3 page).
enter +line	Make the current line the top one.
enter n +line	Move window down n lines.
-line	Move window up (about 1/3 page).
enter -line	Make the current line the bottom one (if possible).
enter n -line	Move window up n lines.
open	Open up one blank line.
enter open	Split the line exactly at cursor position.
enter n open	Open up for n blank lines.
enter ↑↓←→ open	Insert blank lines or rectangle in area defined by cursor.
+page	Move window down one page (page = size of window).
enter n +page	Move down n pages.
-page	Move window up one page.
enter n -page	Move up n pages.
pick	Put the current line into the PICK buffer.
enter n pick	Put n lines into the PICK buffer.
enter ↑↓←→ pick	Place lines or rectangle defined by cursor in PICK buffer.
put	Insert the contents of the PICK buffer (i.e. the lines or rectangle last "picked") at the current position.
enter n put	Insert n copies of the PICK buffer at the current position.
enter ↑↓←→ put	Argument defined by the cursor is used as number of copies.
quote	To put a control character into the file. quote echoes as a ⊕, and whatever key on the keyboard you press next appears as some printable character. The two characters now behave as two characters on the screen, but they are really the single control character in the file. Changing the second letter changes the control character; changing the preceding mark results in two ordinary characters.
refresh	Redraw terminal display.
restore	Insert the contents of the CLOSE buffer (i.e. the lines or rectangle last deleted). Insertion is done at the current position.
enter n restore	Insert n copies of the CLOSE buffer at the current posi-

tion.

enter ↑↓←→ restore	Argument defined by the cursor is used as number of copies.
right	Move window right (about a 1/3 window width).
enter right	Make current column the first one.
enter n right	Move window right n columns.
save	Save the file shown in the current window.
enter name save	Save the file shown in the current window under filename **name**.
enter ↑↓←→ save	Argument defined by the cursor is used as filename.
+search	Search forwards (from the cursor towards the end of the file) for an occurrence of the search key last used.
enter +search	Search key used is from cursor position up to next blank.
enter word +search	Search for the next occurrence of **word**.
enter ↑↓←→ +search	Argument defined by the cursor is used as search key.
-search	Search backwards (from the cursor towards the beginning of the file) for an occurrence of the search key last used.
enter -search	Search key used is from cursor position up to next blank.
enter word -search	Search for the next occurrence of **word**.
enter ↑↓←→ -search	Argument defined by the cursor is used as search key.
use	Switch to the file previously used.
enter use	Edit file, taking its name from cursor position up to next blank.
enter name use	Make the current window look at file **name**. A linenumber and/or searchkey can be specified (as in the *med* command).
enter ↑↓←→ use	Argument defined by the cursor is used as filename.
window	Make another window on the real screen, so that you now have two, or three, or more windows. A "default file" is used. It may be set to look at a file using use , and all other functions work within this little window. If the cursor is on the first or last column of your window the line separating the two windows goes horizontally on the line where the cursor is. The separating line goes vertically if the cursor is on the first or last line of your window. You may have two windows looking at the same file. In fact, it is rather neat, since changes made by editing in either window are reflected (at reasonable intervals) in the other window.
enter window	Delete last created window and return to the previous one.
enter name window	Create another window displaying file **name**.

MACRO

Typing

macro keystrokes macro key

stores the keystrokes sequence into the key. To avoid trouble, key can only be a printable character. From now on pressing the key is equivalent to typing the (long) keystrokes sequence. The sequence

macro macro key

restores the original function of the key.

Miscellaneous

What do the funny characters in the margins mean?

| is normal.
means this is past the end of the file. You may still type stuff - there just wasn't anything there before. Even if you type something on a line, the character won't go away until the line is rewritten by the editor - but the stuff is still there.

< There is still more text to the left (may be blanks).

> There is still more text to the right (not only blanks).

When editing a file for which you don't have write permission, the appropriate editor functions will be disabled.

What to do when disaster comes:

You are protected from loss of files by the insurance system of the MED editor. If you edit a file named *foo*, the old file *foo* is renamed *foo.bak* (the old *foo.bak* is deleted). If you do not like the results of your edit, the UNIX command:

mv foo.bak foo

restores the original file *foo*.

FILES

/tmp/MED* temporary workfile (PICK- and CLOSE-buffer)
$SAVE/MED* saves state of editing session
/usr/lib/med/default default file
*.bak backup files

SEE ALSO

termcap(5), curses(3), keycap(5)

AUTHOR

Dittmar Krall, Wolfratshausen, Germany.
Inspired by the RAND Editor, Steve Zucker e.a., Santa Monica, California.

BUGS

Editor crashes can leave your terminal in a strange state, e.g. with disabled keyboard. Your system administrator should have a command to enable the keyboard. My panic solution is to switch terminal off/on in order to continue.

MED - Tastatur

für

Televideo - 970

Exit	^Z
save to Disk	^D
-Tab	Linefeed
Refresh	^X
ChTab	^T
ChWin	^C
Left	^L
Right	^R
Window	^W
Control char	^\
Macrofunction	^F

<table>
<tr><td>Anfang
Bildschirm
Home
CHAR DELETE</td><td>Suche
vorwärts
+Srch
LINE DELETE</td><td>Seite(n)
vorwärts
+Page
PAGE ERASE</td><td>Zeilen
vorwärts
+Line
PAGE</td><td>

RESET</td></tr>
<tr><td rowspan="2">Tab</td><td>Suche
rückwärts
-Srch
7</td><td>Seite(n)
rückwärts
-Page
8</td><td>Zeilen
rückwärts
-Line
9</td><td>Gehe nach

Goto
-</td></tr>
<tr><td>Speicher

Pick
4</td><td>Füge
Zeile ein
Open
5</td><td>Führe aus

Do
6</td><td>Editiere
andere Datei
Use
9</td></tr>
<tr><td>CE</td><td>Gebe
Speicher aus
Put
1</td><td>Lösche
Zeile
Close
2</td><td>Löschtext
Speicher
Restor
3</td><td rowspan="2">Enter</td></tr>
<tr><td colspan="2">Lösche Zeichen
Delete Char
0</td><td>00</td><td>Füge ein
Insert
.</td></tr>
</table>

MED - Tastatur
für
DSG101

Exit	^Z
save to Disk	^D
-Tab	Linefeed
Refresh	^X
ChTab	^T
ChWin	^C
Left	^L
Right	^R
Window	^W
Control char	^\
Macrofunction	^F

<table>
<tr><td>Anfang
Bildschirm
Home
PF1</td><td>Seite(n)
vorwärts
+Page
PF2</td><td>Zeilen
vorwärts
+Line
PF3</td><td>Suche
vorwärts
+Srch
PF4</td></tr>
<tr><td>Gehe nach
Goto
7</td><td>Seite(n)
rückwärts
-Page
8</td><td>Zeilen
rückwärts
-Line
9</td><td>Suche
rückwärts
-Srch
-</td></tr>
<tr><td>Speicher
Pick
4</td><td>Füge
Zeile ein
Open
5</td><td>Führe aus
Do
6</td><td>Editiere
andere Datei
Use
,</td></tr>
<tr><td>Gebe
Speicher aus
Put
1</td><td>Lösche
Zeile
Close
2</td><td>Löschtext
Speicher
Restor
3</td><td rowspan="2">Enter</td></tr>
<tr><td colspan="2">Lösche Zeichen
Delete Char
0</td><td>Füge ein
Insert
.</td></tr>
</table>

MUNIX - PASCAL
Package

Pascal 68000 User's Guide

The Pascal 68000 User's Guide is intended for use in developing new Pascal programs, and in compiling and executing existing Pascal programs on 68000 Unix systems. This manual gives also some insight into the Pascal 68000 System structure, its components and its behaviour.

Pascal 68000 is an extended implementation of the Pascal language. Specifically Pascal 68000 complies almost completely with the requirements of the ISO standard proposal for Pascal.

This manual is designed for programmers who have a working knowledge of Pascal. Detailed knowledge of 68000 UNIX is helpful but not essential.

Author's initials: DK

Trademarks:

MUNIX, CADMUS	for PCS
DEC, PDP	for DEC
UNIX	for Bell Laboratories

TABLE OF CONTENTS

1. Introduction

The Pascal 68000 User's Guide is intended for use in developing new Pascal programs and in compiling and executing existing Pascal programs on 68000 UNIX systems. This manual also gives some insight into the Pascal 68000 System structure, its components and its behaviour.

This manual is designed for programmers who have a working knowledge of Pascal. Detailed knowledge of 68000 UNIX is helpful but not essential. In any case, it is advisable to get familiar with the UNIX documentation and UNIX standards. The user of this manual should also read the QU68000 documents [1] and [2].

Pascal was designed by Professor N. Wirth as a language for teaching structured programming techniques and as such is used widely in educational institutions. It has also gained popularity as a general-purpose language because it contains a set of language features that make it suitable for many different programming applications. Pascal has furthermore strongly influenced the development of several languages (e.g. ADA). The Pascal language includes a variety of control statements, data types, and predefined procedures and functions.

☛ Throughout this document the following notation is used:

Keywords and **predefined** identifiers are printed in bold face.

Syntactic variables as well as *UNIX components* are printed in italic font.

Metasymbols {,} and [,] are used for optional parts (0..∞ and 0..1), (,) bracket syntactical units and /'separates alternatives. Three dots (...) means a repetition of the preceeding item. Terminal symbols are printed in roman face; to distinguish metasymbols from terminal symbols apostrophes ' are used if necessary.

2. Summary Release 2

2.1. Release 2.1

- Pascal 68000 release 2.1 meets the suggested Pascal ISO Standard [3] more closely than release 1.1 did:

(1) Type **real** is implemented.

(2) Conformant arrays are implemented.

(3) **dispose** is available; alternately, **mark** and **release** can be used.

(4) The attribute **packed** is implemented for subranges. Packing is done on byte level but not on bit level.

- Release 2.1 has some further extensions.

(1) A type **string** is predefined and supported by a few operations.

(2) The attribute **packed** can be applied to simple types.

(3) Besides type **short** there is a type **cardinal** (unsigned) predefined and supported.

(4) There are new standard procedures **pclose** and **pseek**.

(5) **reset** and **rewrite** are extended by a optional second parameter (UNIX filename).

(6) **read** allows variables of type **string**.

- Bits & Pieces

(1) A new option **v**: check arithmetic overflow.

(2) Naming conventions: the compiler prefixes all imported/exported identifiers by an underscore "_" to make the calling of C routines a bit more comfortable.

(3) Some minor, but useful and often used predefined procedures and functions.

(4) Initialisation of the compiler (pass1) is partly done by reading a file "init.h", that is supposed to reside in the directory /usr/include/pc.

(5) Procedures at the outermost level in the main program are exported.

2.2. Release 2.2

- Within modules, variables can be declared at the outermost level. They will exist during the whole program, but cannot be accessed from outside (static variables).

- The size of a procedure is no longer limited.

3. Supported Language

Pascal 68000 is an extended implementation of the Pascal language [4]. Specifically Pascal 68000 adheres to the Pascal language as described in the suggested ISO Standard. This "draft Standard" is a cleaned up version of the original Pascal and has been submitted to ISO for acceptance.

Pascal 68000 release 2.2 has not yet been tested against a "Pascal Processor Validation Suite".

3.1. Deviations

Pascal 68000 deviates from the standard proposal in the following ways:

(1) Only the first 16 characters of an identifier are significant. The truncation of an identifier to 16 characters alters the meaning of a conforming program.

(2) Standard procedures and functions are not allowed as parameters, as in previous standard proposals. Identical results with minor loss in performance can be obtained by declaring user procedures. For example:

```
function userodd(i: integer)  boolean;
    begin
    userodd := odd(i)
    end;
```

(3) Procedures that are to be used as parameters must be declared at main level.

(4) **files** are not allowed in structured data.

(5) Assignment to **for** control variable is done after evaluation of initial expression.

(6) The reserved word **nil** may be redefined.

(7) A **goto** between branches of a statement is permitted.

(8) For **text**files, no final end-of-line is supplied unless requested explicitely by the user.

(9) If the access to record variable in a **with**-statement involves only dereferencing of a pointer variable, this is not done before the statement is executed but for every access to the record.

(10) A null string is accepted by the compiler.

3.2. Extensions

3.2.1. Separate compilation

Pascal 68000 is able to compile so called **modules**, a collection of declarations, procedures and functions. The result of the compilation (an a.out object module) can be handled by the usual UNIX components, i.e. they can be stored in libraries, bound(loaded) with other a.out modules, etc.

The separate compilation feature is further supported by enabling import and export of variables, procedures and functions. Modules implemented in Pascal, C or assembler can be linked to Pascal 68000 modules. Procedures and functions are imported by using a directive in the heading: **extern** for Pascal- and assembler- and **externc** for C-procedures; they are exported implicitely by being defined on the outermost level within a compilation unit. The user is responsible for parameter and result compatibility.

Variables are imported or exported by using the newly introduced keywords **import** or **export** instead of **var**. The predefined variables **input** and **output** are (per default) exported from a mainprogram, if they are listed in the program heading. If used in a module, they have to be imported and must be mentioned in read/write statements. It is not possible to import or export labels!

The new syntax for a compilation unit is:

compilation_unit =
    program *name* '(' *files* ')' ; *block* .
    / **module** *name* ; {*declaration*}

block =
    {*declaration*} *compound_statement*

3.2.2. Additional standard procedures

The following additional standard procedures are available:

addr
This function returns the address of the parameter, which is compatible with all pointer types.

convert
convert (value, typename) returns its first argument as having **type** typename.

☛ No run-time widening is done; e.g.
convert (apointer, anotherpointertype) works, but
convert ('A', **integer**) won't work!

mark, release
mark and release allow to use the heap as a stack. mark(p) stores

the current value of the heap pointer in p. release(p) restores the heap pointer to p. Within one program either **dispose** or **mark** and **release** can be used, but not both simultaneously.

☛ New & dispose are realised by *malloc(3)* and *free(3)* which implement a rather straight-forward memory management. Extensive use of heap will therefor result in remarkable run time penalties. To avoid this problem, use new with mark & release which use a simple and fast memory allocation scheme (see *pc(1)*).

pclose, pseek
They simply supply a Pascal interface to the UNIX system calls *close* and *lseek*.

errorexit
Exit a program and force a core dump.

message
Write a string to the UNIX file *stderr*.

itoa, atoi
Convert **integer** to ascii and vice versa.

date, ptime
Get date and time in ascii representation.

clock
Get cpu time used by the current process.

3.2.3. Strings

String variables are unique to Pascal 68000. Essentially, they are of type **packed array of char** with a dynamic 'length' attribute. The actual length of a **string** is determined by a final zero-byte. **string** variables are not compatible with variables of type **packed array [...] of char**. No range checking is done for **string** operations.

The default maximum length of a **string** variable is 80 characters. This value can be overridden in the declaration by appending the desired length enclosed by []

```
var s  : string;        { 81 bytes will be allocated }
var s1: string [17];    { 18 bytes will be allocated }
```

A **string** variable has a maximum length of 255 characters.

Assignment to a **string** variable can be performed using the assignment statement, the **read** standard procedure or some other routine (e.g. a C standard function). **strings** can be compared, no matter what the current lengths are. Furthermore, it is possible to access the components of a **string** variable; the first one has index 0.

When a **string** variable is used as parameter to **read** or **readln**, all characters up to, but not including, the end-of-line character in the input file will be assigned to it.

When a **string** is written without specifying the field width, the actual number of characters written is equal to the dynamic **string** length. If the field width is longer than the dynamic length, leading blanks are inserted. If the field width is smaller, the **string** is truncated on the right.

Constant strings ('hugo') are compatible with type **packed array** [1..n] **of char** (where n is equal to the string length) and with type **string**. They are also terminated by a zero byte that is not included in the length computation and are limited to a length of 255 characters.

☛ Remember that constant 'a' is a character and not a string. Strings of length one must be supplied by other means (e.g. a C routine).

3.2.4. Generic pointers

Generic pointers provide a tool for generalized pointer handling. Variables of type **address** can be used in the same manner as any other pointer variable with the following exceptions:

generic pointers cannot be deferenced since there is no type associated with them.

generic pointers cannot be used as an argument to **new**.

any pointer can be assigned to a generic pointer. Use **convert** for assigning a generic pointer to a typed pointer.

3.2.5. Attribute packed

As an extension, the attribute **packed** can be applied also to simple types:

```
type byte = packed 0 .. 255;
{ variables of type byte will be allocated one byte }
```

Packed subranges that fit in the ranges -2^7 2^7-1 or 0 2^8-1 are represented in one byte; those fitting in the ranges -2^{15} $2^{15}-1$ or 0 $2^{16}-1$ are implemented in one word (two bytes).

This feature is supported by one-byte, two-byte and four-byte signed and unsigned arithmetic. But the user should keep in mind, that in most cases the packed data have to be extended to a full **integer** entity. See appendix for details.

Types **short** and **cardinal** are defined as

```
type
  short    = packed -32768 .. 32767;
  cardinal = packed 0   65535;
```

3.2.6. Default case

In a case statement a default case can be defined. Either **otherwise** or **else**: will be accepted.

3.2.7. Declaration

The order of declaration for labels, constants, types, variables, functions and procedures has been relaxed. Any order and any number of times declaration sections may be used. Furthermore, **import** and **export** variable declaration are implemented to support the separate compilation feature.

An identifier must be declared before it is used. Two exceptions exist to this rule:

(1) Pointer types may be forward referenced as long as the declaration occurs within the same type-definition-part

(2) Functions and procedures may be predeclared with a forward declaration.

The new syntax for a block is:

block = {*declaration*} *compound_statement*
declaration =
import (*names* : *type*) ;
/ **export** (*names* : *type*) ; ... ;
/ **var** (*names* : *type*) ;
/ **label** *label* , ;
/ **const** (*name* '=' *constant*) ;
/ **type** (*name* '=' *type*) ; ... ;
/ *function_declaration* ;
/ *procedure_declaration* ;

3.2.8. Underscore as letter

The character '_' is significant and can be used in forming identifiers.

3.2.9. Alternate symbols

There are two representations for comment symbols ('(*' '*)' and '{' '}') and for bracket symbols ('(.' , '.)' and '[' , ']').

3.2.10. Exponent

A lower case **e** may be used to indicate real numbers.

3.2.11. Hexadecimal constants

Hexadecimal integers are indicated by a preceding "#". The syntax for a hexadecimal integer is:

unsigned_number = *digit* {*digit*} / # *hexadigit* {*hexadigit*}
digit = 0/1/2/3/4/5/6/7/8/9
hexadigit = *digit* /A/B/C/D/E/F/a/b/c/d/e/f

3.2.12. Character constants

Certain non-printable characters may be represented according to the following table of escape sequences:

\\ backslash
\b backspace
\f form feed
\n newline
\r carriage return
\t horizontal tabulator
ddd

The escape sequence *ddd* consists of a backslash followed by 1,2, or 3 octal digits which are taken to specify the value of the desired character. If the character following the backslash is not one of those specified, the backslash is ignored.

3.2.13. Additional predefined identifiers

See also /usr/include/pc/init.h

```
const
    minint      = -maxint - 1;
    maxshort    = 32767;
    maxcard     = 65535;
    maxchar     = '\377';
    minshort    = -32768;
    mincard     = 0;
    minchar     = '\000';
type
    alfa   = packed array [1..16] of char;
    short = packed minshort .. maxshort;
    cardinal    = packed mincard .. maxcard;
    {address is predefined too}
var
    argc  : integer;
    argv  : ^ array [1..100] of ^ string;
    environ     : ^ array [1..100] of ^ string;

function   pclose (var f: file_of_any_type): integer; externc;
function   pseek (var f: file_of_any_type; offset: integer;
                  whence: short): integer; externc;
procedure message (s: string); externc;
procedure itoa  (i: integer; var s: string); externc;
function   atoi  (s: string): integer; externc;
```

```
procedure date (var datevar: string); extern;
procedure ptime (var timevar: string); extern;
function  clock : integer; extern;
```

3.3. Implementationdefined

(1) **maxint** is 2 147 483 647.

(2) **set** bounds are 0 and 255.

(3) A variable is selected before the expression is evaluated in an assignment statement.

(4) Default field width specification:
12 for **integers**, 12 for **reals** and 5 for **booleans**.

(5) **reset** on the standard input file resp. **rewrite** on the standard output file is permissible.

3.4. Error Handling

(1) Uninitialized or undefined variables are not detected.

(2) A missing **reset** or **rewrite** statement is not detected (see 4.3).

(3) No runtime checks are performed on the tag field of variant records.

(4) No bounds checking is performed on overlapping set operands.

(5) The use of a function without an assignment to the function value variable is permitted.

(6) No checks are inserted to check pointers after they have been assigned a value using the variant form of new.

(7) No bound checks are inserted for the **succ**, **pred** or **chr** functions.

(8) The evaluation of expressions involving big constants can cause compiler crash.

(9) The **for** control variable is not invalid after the execution of the **for** statement.

(10) Two nested **for** statements with the same control variable are permitted, but do not run into an infinite loop.

(11) Type declarations of the kind

```
type t = type1;


procedure p;
type
   pt = ^ t;
   t  = type2;
```

are not correctly analysed.

4. Pascal 68000 under UNIX

4.1. Creating and executing a program

The usual way to create and execute a program is realized by entering the following commands to the 68000 UNIX operating system. With each command, you include information that further defines what you want the system to do. Of prime importance is the file specification, which indicates the file to be processed. You can also specify qualifiers that modify the processing performed by the system ($ is the system prompting symbol).

A program is entered or corrected by any editor of the user's taste. The file name of Pascal source programs must have the suffix '.p'.

```
$ edit <name>.p CR
```

The *pc* [5] command compiles and links the Pascal program. The resulting object module is left in <name>.

```
$ pc -o <name> <name>.p CR
```

The program is loaded and run by:

```
$ name CR
```

The only program parameters supported by the Pascal language are files. There are three ways to associate an (external) UNIX file specification with an (internal) Pascal file specification. The standard Pascal files **input** and **output** are always associated with the logical UNIX files *stdin* and *stdout*. Their comfortable and flexible use is described in [6]. All other Pascal files can be associated with any UNIX file either by assignment within a commandline:

```
$ name pascalfile_1=UNIX_file_specification  \
       pascalfile_2=UNIX_file_specification [1] \
           CR
```

or - if an assignment is missing - interactively:

```
$ name pascalfile_2=UNIX_file_specification CR
pascalfile_1 ? UNIX_file_specification CR
```

[1] As these assignments are considered as one argument each, there must be no blanks before or after the '=' sign.

Example:
$ ed example.p CR
$ pc -o example example.p CR
$ example eingabe=/dev/tty CR
ausgabe ? example.aus CR

It is advisable to define a shell procedure for a more convenient file assignment especially if some of the files are "fixed" or work files.

As an extension of Pascal 68000, the external filename can be stated in the corresponding **reset** or **rewrite** statement. In this case, a file assignment at run time will be meaningless.

For efficiency, output is in general buffered; the buffer is flushed only when it's full, or - in case of **text** files - if a **writeln** is stated. To facilitate interactive I/O, output to a **file of char** will be unbuffered, if the file is connected to a terminal (*/dev/tty*). Input from a terminal is line-oriented unless the terminal is in raw mode (see *stty(1)*, *ioctl(2)*).

4.2. Support of UNIX Facilities

The user has full access to all UNIX system calls as well as files. The system calls can be accessed just like C procedures (see below). The objects are to be found in the standard library. The predefined procedure **halt** provides a means to return an "exit code" to the system. Furthermore the "system variables" argc, argv and environ are provided for access to the command arguments and the process environment. argc indicates the number of arguments whereas argv references an array of argument strings. The actual length "i" of environ is determined by environ^[i] = **nil**. See 3.2.13 for a declaration of these variables.

Remember that the file specifications are command arguments too, i.e. argv^[1]^[0] is the first character of the program name. If you access argv^ or environ^, it is essential to switch off the pointertest.

Example:

The following statement will print out the environment:

```
{$t- }
begin
i := 1;
while environ^ [i] <> nil do
   begin
   writeln(output, environ^[i]^);
   i := i+1
   end
end;
```

The C preprocessor *cpp* [7] may be used without limitations.

As stated earlier C and assembler modules can be loaded with Pascal. When connecting Pascal modules with C modules you should take care of parameter compatibility (see 9.). Moreover, you must be sure that the C modules do not use the *sbrk/brk* system call which interferes with the Pascal heap.

4.3. Limitations of Pascal 68000

Because of the separate compilation feature, a missing **reset** or **rewrite** cannot be detected by the compiler and will most probably cause the program to crash with an address error at the first attempt to access the corresponding file-variable.

In general, it is possible to **reset input** or **rewrite output** with the effect, that *stdin* or *stdout* is repositioned to the beginning of the associated file. If the file is connected to a pipe this will have no effect at all.

Because of a very straight-forward register allocation mechanism, there may be very deeply nested expressions that do not compile.

5. Compiler options

Compiler options are given by using "{$...}". Each option consists of a lower or upper case letter followed by "+" or "-". Options are separated by commas. There must be no blanks between "{" and "$" or between a comma and the succeeding option. The following options are supported:

```
A/a +/-  a+ produces an assembler listing
B/b +/-  b+ accept C-string notation (i.e. with backslash)
D/d +/-  d+ produces code for pointer, subrange and arithmetic overflow
             check and the tracing of line numbers
E/e +/-  e- will suppress extension warnings
P/p +/-  p+ code for profiling is generated (see prof [8])
T/t +/-  t+ produces code for pointer check
V/v +/-  v+ produces code for arithmetic overflow check
W/w +/-  w- will suppress warning messages

Defaults: {$a-,b+,d+,e-,w+}
          d+ implies v+ and t+.
          d - implies v - and t -.
```

Options appearing before the **program** or **module** symbol can be overwritten by options given in the *pc* command.

6. Error handling

Errors are detected and reported by several components of the Pascal 68000 system:

- preprocessor *cpp*, see *pc*
- compiler
- loader *ld*, see *pc*
- run-time system

6.1. Compiletime Detection of Source Errors

pass1 detects syntactical and some semantical errors and (optionally) deviations from the standard language definition. Errors that are not detected are listed in 3.4. *pass1* does not produce error messages or a listing but compiles a condensed version of the diagnostics that will be printed in a readable form by an extra pass *perror*. If only warnings are issued compilation proceeds otherwise it will be terminated.

The option "-L" (see *pc*) produces a full listing with error messages. The default is a listing containing the offending line, its predecessor and the readable (and hopefully understandable) error messages. Sometimes an error causes several messages to be printed in which case all but the first one can be ignored.

pass2 detects no source errors, but might report a violation of a compiler limitation (see 4.3) or a compiler error (though, of course, it should not). Please let us know if you get an compiler error or an unknown error message.

6.2. Other Errors Detected at Compiletime

During compilation, UNIX resources can be exhausted, e.g. file system, process table, or memory overflow, etc. Furthermore UNIX can deny access to files. Extensive treatment of these errors is beyond the scope of this manual. They are described in the UNIX documentation.

6.3. Runtime errors

Errors occurring at run time are always fatal, i.e. the program will report an error message, dump the core file and then abort. With the help of the UNIX debugger *adb* [9] the user can generate a (partially) symbolic post mortem dump which indicates the location of the fatal error, the number of the corresponding source line (if the debug option was on), the dynamic calling sequence, etc.

The error message should comprise a sufficient diagnosis of the error detected. As far as file access is concerned, the errors are mostly

reported by the UNIX system and must be investigated using the documentation. Other messages indicate programming errors such as divide by zero, integer overflow , etc.

7. Pascal System Components

Figure 7.1 gives an overview of the Pascal system components. The preprocessor is the same as the C-compiler's, and does macro preprocessing and including of source files.

The first pass *pass1* does the lexical, the syntactical and the semantical analysis. It constructs a parse tree for each block and outputs it.

There is an extra pass *perror* to produce a source listing if requsted, and to print error messages based on the diagnostics compiled by *pass1*.

The second pass *pass2* does the code generation. The output of *pass1* is read in and an identical parse tree as in *pass1* is built up. The generated code is output in several files.

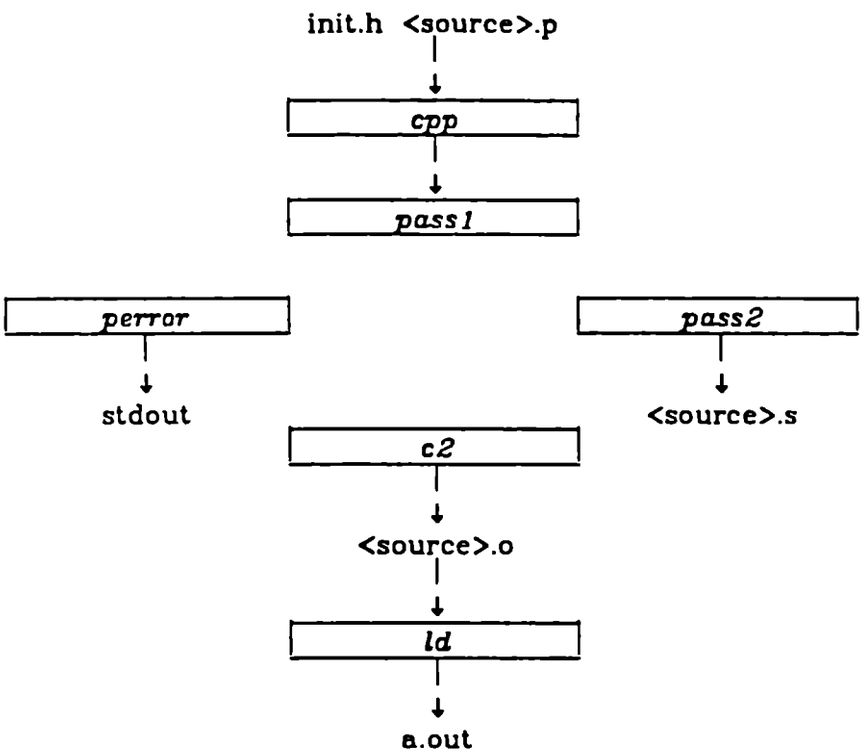


Figure 7.1: Pascal System Components

The third pass *c2* is the same as the C-Compiler's and collects the code distributed on various files. It produces one file containing the object code in *a.out* format [10].

7.1. Hardware and Software Environment

Pascal 68000 runs on two quite similar 68000 UNIX systems, PERKEO [11] and QU68000 [12].

Features of these systems are:

actually supported microprocessor Motorola 68000

large memory (≥ 0.5 MB)

hard disk (≥ 10MB)

memory management unit

UNIX [13] [14] is a time-sharing system and provides all such features needed by Pascal 68000. Of great importance are the file system, the command interpreter (shell), and the object module management (*ar*, *ld* [15]). But of equal, if not greater, weight are all those UNIX components which make life easier: *ed*, *med*, *make*, etc.

7.2. Pass1

The first pass does lexical analysis, parsing, declaration handling, tree building, and some optimization. This pass is largely machine independent.

The lexical analysis is a conceptually simple procedure that reads the input and returns the tokens of the Pascal language as it encounters them: identifiers, constants, operators and keywords. Comments must be skipped. Decimal and hexadecimal constants, characters and strings must be properly dealed with.

The first pass parses, as the original Pascal-P4 compiler, the tokens in a top down, left right, recursive descendent fashion. During the processing of a declaration part a symbol table is built up, addresses will be allocated to variables and procedures, and the semantic of the declaration is checked. Besides this, information is prepared for the debugger *adb*.

During the processing of the statement part a parse tree is built. The proper use of operators and operands is checked. Some complex syntactical structures are broken down into simpler ones.

7.3. Pass2

The second pass generates 68000 machine code and related information from the source program represented by the parse tree; these will be combined to a *a.out* object. In addition it completes the debugger information prepared by *pass1*.

Code generation is done in two steps. The first one traverses the parse tree generating pseudo instructions for a hypothetical stack machine. This step is rather simple, machine independent and straightforward. The second step deals with machine specific tasks such as address computation, register allocation and 68000 code generation, thus interpreting the pseudo instructions in a real existing environment. Step one and two take place in parallel for efficiency reasons; the generation of a pseudo instruction is replaced by calling a procedure implementing this instruction.

pass2 yields different kinds of output. Three files contain binary information ready to be combined to an *a.out* object by a third pass. A fourth file may contain the generated machine code in an assembler-like representation. On another file debugger information is prepared. And there might be diagnostic messages indicating a violation of a limitation or a compiler error.

7.4. Cross Reference

The cross reference program *xref* produces the usual cross-reference list with identifiers and line numbers.

xref is implemented as an independent component. There are several reasons for doing so:

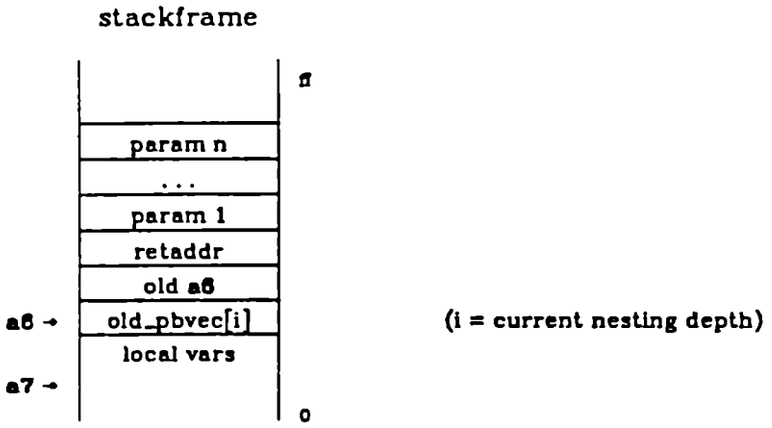
The compiler is not bothered with cross referencing.

The multiprogramming (parallel processing) of the system can be exploited.

Many programming errors can be found with the help of a cross-reference listing; i.e. it is not necessary to start the big, resource consuming, compiler.

8. Calling Conventions

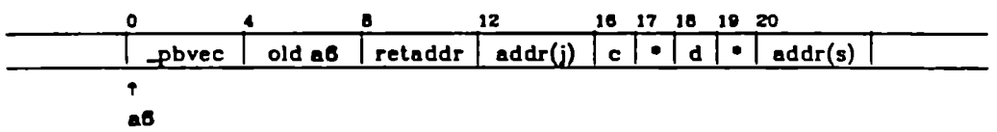
Procedure calls are realised by a commonly used mechanism defining stackframes, which are allocated or deallocated as a procedure is activated or deactivated, respectively. In our implementation four registers and a vector of pseudo registers are used: **a7** is used as stackpointer, **a6** and **a5** reference the current and the global stackframe base, respectively. **d0** returns a function result and the system variable _pbvec[0..maxdepth] stores the base addresses of all currently accessible stackframes. The layout of a stackframe is shown below.



A parameter is passed by-reference or by-value depending on whether it was declared as **var** parameter or not. For long parameters (> 4 bytes; **strings**) the parameter address is transferred and - in case of value parameters - the parameter itself copied within the procedure body. The representation of parameters in memory is the same as for other variables with the exception that they are always word aligned, i.e. a parameter occupying an odd number of bytes in memory will always be followed by a byte of undefined storage.

Example:

stackframe for
procedure p(**var** j:**integer**; c,d:**char**; s:**string**);
after procedure entry

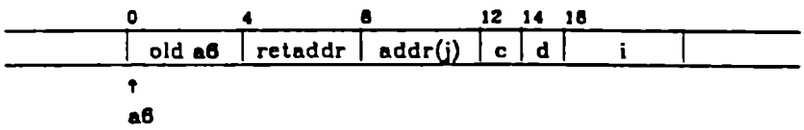


• undefined

The C interface provided as an extension differs from the Pascal parameter passing conventions only in the treatment of one-byte values; following the C semantics, data occupying one byte of storage are extended to (unsigned) word values and then treated like a **short**.

Example:

stackframe for
procedure p(**var** j:**integer**; c,d:**char**; i:**integer**); externc;
after procedure entry



where c & d are pushed as zero extended **short** items
(at our installation Pascal **short** corresponds with C **int**).

9. Data Representation and Allocation

In a.out modules program data fall into three segments: the text segment, the data segment and the bss segment. Pascal 68000 uses only two of them: the text segment contains program code and constants whereas static data (variables declared at the outmost level in a module) and **exported** variables are stored in the bss segment. Automatic and dynamic data are allocated at runtime in stack and 'free memory' managed by the *brk/sbrk* system calls, respectively.

To cope with alignment, one general rule can be stated: any data allocated more than one byte is aligned on a word (2 bytes) boundary.

Variables of scalar and pointer types are allocated storage space as summarized in table 9.1. Variables of subrange types are allocated as variables of the associated scalar types. For example, a type 1..10 is considered a subrange of **integer** and therefor allocated a longword. Variables of **packed** subrange types are implemented in one byte (-128 .. 127; 0 255), two bytes (-32768 .. 32767; 0 .. 65535) or four bytes (all others). The structured types are stored as described below.

Type	Storage	Allocation
character	8 bits (1 byte)	Byte
boolean	8 bits (1 byte)	Byte
short	16 bits (1 word)	Word
integer	32 bits (1 longword)	Word
real	32 bits (1 longword)	Word
enumerated	8 bits (1 byte) if type contains 256 elements or less; 16 bits (1 word) otherwise	Byte if type contains 256 elements or less; Word otherwise
pointer	32 bits (1 longword)	Word

Table 9.1: Storage of Scalar and Pointer Types

A **set** is allocated storage depending on the ordinal value of its largest element: the number of bytes it occupies is equal to the ordinal value rounded up to the nearest byte boundary. Since the size of a **set** is limited to 256

elements, with ordinal values from 0 to 255, a **set** occupies at most 32 bytes.

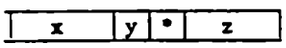
An **array** is stored and aligned according to the type of its components. For example, each element of a character **array** is stored in a byte and aligned on a byte boundary; if the **array** has more than one component then the total **array** is aligned a on word boundary. Similarly, each element of an **array of set of** 3..21 occupies three bytes and is aligned on a word boundary.

strings and constant strings are internally terminated by a binary 0; i.e. they adhere to the C convention.

Records are stored and aligned field by field according to the type of the field. For example, a variable of type

```
record
  x: integer;
  y: boolean;
  z: 1..10
  end;
```

is aligned on a word boundary because it occupies more than one byte of storage. The figure beneath shows how this variable is stored:

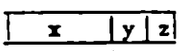


• undefined

The attribute **packed** does not affect the allocation of data structures; it is only interpreted with subranges as stated earlier. To yield a more compact representation than in the example above, you had to define the following structure:

```
record
  x: integer;
  y: boolean;
  z: packed 1..10
  end;
```

which would result in the following storage allocation:



10. Appendix

10.1. Examples

10.1.1. Sample program

```
 1 program ackermann(input,output);
 2  var x, y: integer;
 3  function ack(i,j: integer): integer;
 4   begin
 5   if  i = 0 then  ack:=j+1
 6   else  if   j = 0 then  ack:=ack(i-1,1)
 7         else      ack:=ack(i-1,ack(i,j-1))
 8   end;
 9
10  begin
11   repeat
12    writeln(output,'Enter two integers. Terminate with zero.');
13    read(input,x,y);
14    writeln(output,'acker(',x:0,',',y:0,') = ',ack(x,y):0)
15    until x=0
16  end.
```

10.1.2. Crossreference

ack	3p	5	6	6	7	7	7	14
ackermann	1							
i	3f	5	6	7	7			
input	1	13						
integer	2	3	3					
j	3f	5	6	7				
output	1	12	14					
read	13							
x	2v	13	14	14	15			
y	2v	13	14	14				

10.1.3. Separate Compilation

```
 1 program ackermann(input,output);
 2 var x, y: integer;
 3
 4 import counter: integer;
 5
 6 function ack(i,j: integer): integer;  extern;
 7
 8 begin
 9  repeat
10    writeln(output,'Enter two integers. Terminate with zero.');
11    read(input,x,y);
12    counter:=0;
13    writeln(output,'acker(',x:0,',',y:0,') = ',ack(x,y):0);
14    writeln(output,'number of calls=',counter:5);
15  until x=0
16 end.
```

```
 1 module ack;
 2
 3 export counter: integer;
 4
 5 function ack(i,j: integer): integer;
 6  begin
 7  counter:=counter+1;
 8  if    i = 0 then  ack:=j+1
 9  else  if    j = 0 then  ack:=ack(i-1,1)
10          else       ack:=ack(i-1,ack(i,j-1))
11  end;
12
```

10.2. Coercions

The following gives an overview about the coercions implemented by Pascal 68000 to provide for consistent use of one-, two- and four-byte signed and unsigned operands. The rules are valid for all arithmetic and relational operations.

Index expressions are always extended to a full integer value. Expressions representing setelements are considered unsigned one-byte.

	f	i	i1	i2	u	u1	u2
f	f	f	f	f	f	f	f
i	f	i	i	i	i	i	i
i1	f	i	i1	i	i	i	i
i2	f	i	i	i2	i	i	i
u	f	i	i	i	u	u	u
u1	f	i	i	i	u	u1	u
u2	f	i	i	i	u	u	u2

Notation

f: floating point
i: (signed) integer
u: unsigned (0 .. 2^{31}-1)
i1: one-byte integer
i1: two-byte integer
u1: one-byte unsigned
u2: two-byte unsigned

10.3. Standard Procedures and Functions

For a detailed description see [3], [4].

Procedure	*Parameter*	*Result*	*Function*
abs(x)	**integer** or **real**	same as x	Computes the absolute value of x.
addr(x)	any type	**address**	Type address is compatible with all pointer types.
arctan(x)	**integer** or **real**	**real**	Computes the arcus tangens of x.
atoi(s)	**string**	**integer**	Convert ascii string to **integer**.
chr(x)	**integer**	**char**	Returns the character whose ordinal number is x.
clock		**integer**	Returns the cpu time (in milliseconds) used by the current process.
convert(a,t)	any type	t	Returns value of a with type t.
cos(x)	**integer** or **real**	**real**	Computes the cosinus of x.
date(x)	**string**		Return date in ascii representation: dd-mon-yyyy
eof(f)	file	**boolean**	End of file encountered. **true** only when the file position is after the last element in the file.
eoln(f)	file	**boolean**	End of line encountered. **true** only when the file position is after the last character in a line. The value of f^ is a space.
errorexit			Exit program and force a core dump.

exp(x)	**integer** or **real**	**real**	Computes the base of the natural logarithmus raised to the power of x.
get(f)	file		Moves the current file position to the next element.
halt(x)	**integer**		Terminates the execution of the program and returns the value of x. See also the system call **exit**(2).
halt			same as **halt**(0).
itoa(i,s)	**integer**,**string**		Convert **integer** to ascii; the result is delivered in variable s.
ln(x)	**integer** or **real**	**real**	Computes the natural logarithmus of x.
mark(x)	any pointer		Stores the current value of the heap pointer into x.
message(x)	**string**		Write string to *stderr*.
new(p)	any pointer		Allocates heap memory and returns the address in p.
new(p,t1,..tn)			Allocates heap memory to pointer. p^ must be a record with variants.
odd(x)	**integer**	**boolean**	Returns **true** if the integer x is odd; **false** otherwise.
ord(x)	any scalar type except **real**.	**integer**	Returns the ordinal (integer) value corresponding to the value of x.
pack(a,i,z)			z:=a[i..n]; where i..n: index range of z.
page(f)	file		skip to the top of a new page before printing the next line of the textfile f. The default for f is **output**.
pclose(f)	file		See system call close (2).

pred(x)	any scalar type except **real**	same as x	Returns the predecessor value of x.
pseek(f,o,w)	file, **integer**, **short**		See system call lseek (2).
ptime(x)	**string**		Return time in ascii representation: hh-mm-ss.0
put(f)	file		Appends f^ to the file f. The default for f is **output**.
read(f,x)	file type of x depends on the filetype		Reads the value of x from the file f. The default for f is **input**.
readln(f)	**text**		Skips to the beginning of the next line. The default for f is **input**.
read(f,x1,..,xn)			same as **read**(f,x1); **read**(f,x2,..xn)
readln(f,x1,..,xn)			same as **read**(f,x1,..,xn); **readln**(f)
release(x)	any pointer		Loads the heap pointer with the value of x.
reset(f[,s])	file,**string**		Resets file for reading. The optional second parameter designates a UNIX pathname.
rewrite(f[,s])	file,**string**		Resets file for writing. The optional second parameter designates a UNIX pathname.
round(x)	**real**	**integer**	trunc(x + 0.5) if x >= 0 trunc(x - 0.5) if x < 0
sin(x)	**integer** or **real**	**real**	Compute sinus of x.
sizeof(x)	any type	**integer**	Returns size used to represent variables of same type as x.
sqr(x)	**integer** or **real**	same as x	Computes the square of x.

sqrt(x)	**integer** or **real**	**real**	Compute square root of x.
succ(x)	any scalar type except **real**	same as x	Returns the successor value of x.
trunc(x)	**real**	**integer**	Truncate x to a **integer** value.
unpack(z,a,i)			a[i..n]:=z; where i..n: index range of z.
write(f,x)	**text** type of x depends on the filetype		Writes the value of x to the file f. The default for f is **output**.
writeln(f)	file		Starts a new line. The default for f is **output**.
write(f,x1,..,xn)			same as **write**(f,x1); **write**(f,x2,..xn)
writeln(f,x1,..,xn)			same as **write**(f,x1,..,xn); **writeln**(f)

10.4. Syntax Equations

```
letter =      a/b/c/d/e/f/g/h/i/j/k/l/m/n/o/p/q/r/s/t/u/v/w/x/y/z
       /A/B/C/D/E/F/G/H/I/J/K/L/M/N/O/P/Q/R/S/T
       /U/V/W/X/Y/Z/_

digit = 0/1/2/3/4/5/6/7/8/9

name =              letter { letter / digit }
constant_name =     name
type_name =         name
variable_name =     name

names =             name , ...
label =             unsigned_number

compilation_unit =
  program '(' names ')' ; block .
  / module name ; {declaration}

block = {declaration} compound_statement

declaration =
  / import ( names : type ) ; ... ;
  / export ( names : type ) ; ... ;
  / var    ( names : type ) ; ... ;
  / label label ,    ;
  / const  ( name '=' constant ) ;
  / type   ( name '=' type ) ; ... ;
  / function_declaration ;
  / procedure_declaration ;

constant =
  [ sign ] ( unsigned_number / constant_name )
  / character_string

sign = + / -

unsigned_constant =
    unsigned_number / character_string
  / constant_name / nil

unsigned_number =
  digit {digit}  / # hexadigit {hexadigit}

hexadigit =         digit /A/B/C/D/E/F/a/b/c/d/e/f
character_string =      /*characters enclose by '' */
```

t y p e ----------

```
type =          type_name / new_type
new_type =      simple_type / structured_type / ^ type_name

simple_type =       ordinal_type / real

ordinal_type =
    '( names ')'
  / constant .. constant
  / ordinal_type_name

structured_type =
    [ packed ] unpacked_structured_type
  / structured_type_name

unpacked_structured_type =
    array '[' ordinal_type , ... ']' of type
  / file of type
  / record [ field_list [ ; ] ] end
  / set of ordinal_type

field_list =
    fixed_part [ ; variant_part ]
  / variant_part

fixed_part = ( names : type ) ; ...
variant_part =      case [ tag_field_name : ] tag_type of variant ;
variant =           case_constant_list : '(' [ field_list [ ; ] ] ')'

case_constant_list =    case_constant , ...
case_constant =         constant [ .. constant ]
```

e x p r e s s i o n ----------

```
variable =    ( variable_name / field_name )
  { '[' expression ,    ']'
   / . field_name
   / ^ }

factor =
    variable
  / unsigned_constant
  / function_name [ '(' actual_parameter , ... ')' ]
  / set
  / '(' expression ')'
  / not factor

actual_parameter =
  expression
  / procedure_name / function_name
```

```
set    = '[' [ member ,    ] ']'
member =    expression [    expression ]

term = factor
  { ( * / '/' / div / mod / and ) factor }

simple_expression = [ sign ] term
  { ( + / - / or ) term }

expression = simple_expression
  [ ( <> / = / < / > / <= / <= ) simple_expression ]


                  p r o c e d u r e ----------

procedure_declaration =
  procedure_heading ;
  ( block / directive )

function_declaration =
  function_heading ;
  ( block / directive )

directive =          forward / extern / externc
procedure_heading =     procedure name [ '(' parameter ;    ')']

function_heading  =     function name ['(' parameter ; ... ')']
              : result_type

parameter =
    function_heading / procedure_heading
  / names : type
  / var names : ( type_name / array_type )

array_type =
  array '[' index_type ;    ']' of (type_name / array_type )

index_type =
  name    name : ordinal_type_name


---------- s t a t e m e n t ----------

statement = [ label : ]
    compound_statement
  / case expression of case_element ; ... [ ; ] end
  / for name := expression
        ( to / downto ) expression
        do statement
  / goto label
  / if expression then statement
        [ else statement ]
```

/ **repeat** *statement* ; **until** *expression*
/ **while** *expression* **do** *statement*
/ **with** *variable* , ... **do** *statement*
/ (*variable* / *function_name*) := *expression*
/ *procedure_name* ['(' *actual_parameter* , ... ')']

compound_statement =
begin *statement* ; ... **end**

case_element =
((*case_constant* / **else**) : / **otherwise**)
statement

10.5. Reserved Identifiers

The following identifiers are predefined by Pascal 68000.

abs	environ	mincard	readln
addr	eof	minchar	real
address	eoln	minint	release
alfa	errorexit	minshort	rewrite
arctan	exp	new	round
argc	false	nil	short
argv	get	odd	sin
atoi	halt	ord	sqr
boolean	integer	pack	sqrt
cardinal	itoa	page	succ
char	ln	pclose	text
chr	mark	pred	true
clock	maxcard	pseek	trunc
convert	maxchar	ptime	unpack
cos	maxint	put	write
date	maxshort	read	writeln
	message		

Furthermore, there are identifiers of data and subroutines used by the Pascal run time system that are referenced at linking time. Among them are system calls (e.g. open, close) and some C standard subroutines; all others are listed below. Warning: the user might use this identifiers to define own procedures or data and the linker ld will reference the users' entity and report no error.

_copen	_pconf	_pperr	_pwrf
_div10	_pemptyset	_prdi	_pwri
_entry	_pemptystr	_prdr	_pwrr
_error	_phigh	_prds	_pwrs
_ferror	_pierr	_pserr	_syserr
_flushbuffer	_pjerr	_pstaticstring	atan
_pblank	_plim	_puerr	closefiles
_pbvec	_plow	_pwrb	final
_pcase	_popen	_pwrc	fsqr
_pcerr	_popenfileindx	_pwrc1	ln
_pclose	_popenfiles		main

11. References

[1] Uhlenberg, M.: *Unterschiede von Munix zu UNIX V7*
Siemens AG, ZTI INF 212

[2] Uhlenberg, M.: *Munix - Erste Schritte auf dem QU68000*
Siemens AG, ZTI INF 212

[3] Addyman, A. M.: *BSI / ISO Working Draft of Standard Pascal*
BSI DPS / 13 / 4 Working Group, Pascal News #14, January 1979

[4] Jensen K.; Wirth N.: *Pascal User Manual and Report*
Second Corrected Reprint of the Second Edition 1978, Springer Verlag

[5] *UNIX Programmer's Manual; pc(1)*

[6] *UNIX Programmer's Manual; Volume 2; 2,3,6,...*

[7] Kernighan B.; Ritchie D.: *The C Programming Language*
Prentice-Hall Software Series; Prentice Hall, Inc.; New Jersey

[8] *UNIX Programmer's Manual; prof (1)*

[9] *UNIX Programmer's Manual; adb (1)* and
UNIX Programmer's Manual; Volume 2; 18

[10] *UNIX Programmer's Manual; a.out (5)*

[11] *PERKEO - a Hardware / Software System for Personal, Scientific Computing*
Siemens AG, ZT ZFE FL AIF

[12] *Q68K - Ein Q-Bus-Prozessor hoher Leistung auf 68000-Basis*
Spezification Version 1; PCS interner Bericht; 1982

[13] *UNIX Time-Sharing System*
The Bell Sytem Technical Journal, July-August 1978, Vol. 57, No.6, Part 2

[14] *UNIX Time-Sharing System*

UNIX Programmer's Manual, Seventh Edition, Volume 1 & 2A, January 1979 Bell Telephone Laboratories, Inc.

[15] *UNIX Programmer's Manual; ar(1), ld(1)*

Berkeley Extension Summary

Trademarks:

MUNIX, CADMUS	for PCS
DEC, PDP	for DEC
UNIX	for Bell Laboratories

The MUNIX extension package consists of utilities of the fourth Berkeley Distribution. This manual contains the command descriptions and supplementary documents.

1. MUNIX Extension Package - Summary

   Features of the extension package.

2. Writing Papers with nroff using -me

   A popular macro package for nroff.

3. -me Reference Manual

   The final word on -me.

4. Writing tools - the Style and Diction Programs

   Description of programs which help you understand and improve your writing style.

5. The Berkeley Font Catalog

   Samples of fonts currently available for the raster plotters.

6. Screen Updating and Cursor Movement Optimization

   Description of a package of C library functions which allow screen updating (with user input) and cursor motion optimization.

7. Command Descriptions in MUNIX Ia ("Berkeley")

   Command descriptions in the style of the MUNIX Programmers' Manual Volume I.

MUNIX Extension Package V1.1 - Summary

PCS
Pfälzer-Wald-Str. 36
D-8000 München 90

The MUNIX Extension Package includes a set of useful utilities from the fourth Berkeley Distribution. Some of them are extensions of UNIX† V7 commands, i.e. cat and man, others are complete new. The most important facility of this package is the screen oriented editor VI.

1. Basic Software

1.1. User Access Control

CHFN — Change full name of user. The gcos field of the passwd-file is used to give additional information about the user like phone number, office...

1.2. Terminal Handling

TSET — Set terminal modes.

RESET — Reset the terminal bits to a sensible state. Useful after a program dies leaving a terminal in a funny state.

CLEAR — Clear terminal screen.

LOCK — Reserve a terminal.

1.3. File Manipulation

CAT — Concatenate and print one or more files onto standard output. Additional options to the CAT command of the 7th edition:

- # Numbering of output lines.
- # Crushing out multiple adjacent empty lines.
- # Printing non-printing characters in a visible way.

SEE — Print a file which contains non-printing characters in a readable format.

MORE — Interactive display function for text files.

- # Start at linenumber i or two lines before pattern.

---

†UNIX is a Trademark of Bell Laboratories.

Display next page or i-more lines.
Skip i lines.
Search i-th occurence of pattern.
Display current filename, current line number.
Define window size.
Squeeze multiple blank lines.
Start up the editor VI at current line.
Exit to Shell.
Help function.

VPR Raster printer/plotter spooler.

HEAD Give first few lines of a stream.

STRINGS Look for ASCII strings in a binary file.

FOLD Fold long lines for finite width output device.

NUM Number lines.

UL Do underlining.

COLRM Remove selected columns from a file.

EXPAND
UNEXPAND Expand tabs to spaces and vice versa.

COMPACT
UNCOMPACT Compress and uncompress files using an adaptive Huffman code.

CCAT Cat the original file from a file compressed by compact, without uncompressing the file.

1.4. Running of Programs

YES Be repetitively affirmative. Outputs 'y'.

1.5. Status Inquiries

FINGER List the current users including login name, terminal name, login time...

W Print a summary of the current activity on the system, including what each user is doing.

USERS Print a compact list of users who are on the system.

UPTIME Show how long system has been up.

1.6. Backup and Maintenance

SHUTDOWN Close down the system at a given time. Used to notify users nicely when the system is shutting down.

BADSECT Create files to contain bad sectors. Less general than bad block forwarding, but better than nothing.

SCRIPT Make a typescript of everything printed on the terminal during a session.

DMESG Look in a system buffer for recently printed diagnostic messages and print them on the standard output.

1.7. Accounting

SA Publish Shell accounting report. Gives usage information on each command executed. Additional options to the SA command of the 7th edition.

- # Number of times used.
- # Total system time, user time and elapsed time.
- # Optional averages and percentages.
- # Sorting by different criterions:
  - disk I/O operation
  - cpu-time average memory usage
  - cpu-storage integral
  - number of calls

LAST Indicate last logins of users, groups or on specified terminals.

LASTCOMM Show last commands executed in reverse order.

1.8. Communication

MSGS Read system messages. These messages are sent by mailing to the login 'msgs'.

BIFF Be notified if mail arrives and who it is from.

FROM Show who is the sender of my mail.

PRMAIL Print the mail which waits for you, or a specified user, in the 'post office'.

LEAVE Remind you when you have to leave.

1.9. Basic Program Development Tools

CURSES Library of screen functions which allows screen updating (with user input) and cursor motion optimization.

WHAT Show what versions of object modules were used to construct a file.

ERROR Analyze and disperse compiler error messages.

- # Knows about error messages produced by MAKE, CC, CPP, AS, LD, LINT, PC, F77.
- # Attempts to determine which language processor produced each error message.
- # Determines source file and line number to which the error message refers.
- # Determines if the error message is to be ignored, or not.
- # Inserts the error message into the source file as comment

PRINTENV Print out the values of the variables in the Shell environment.

1.10. Unix Programmers' Manual

MAN — Give information from the on line programmers' manual.

- # Gives all commands whose description contains any of a specified set of keywords.
- # Attempts to locate manual sections related to a specified list of files.
- # Formats a specified set of manual pages.

CATMAN — Create the preformatted versions of the on-line manuals.

APROPOS — Show which manual sections contain instances of any of the given keywords in their title.

WHATIS — Look up a given command and give the header line from the manual section.

WHEREIS — Locate source, binary, and or manual for specified files.

2. Tape Manipulation

MT — Give commands to the tape drive.

- # Space forward i files or records.
- # Space backward i files or records.
- # Write i end-of-file marks.
- # Rewind tape.
- # Swap or do not swap bytes.

REWIND — Rewind the tape drive.

3. Text Processing

3.1. Document Preparation

EX — The line oriented text editor EX is a superset of the ED editor from UNIX V7 and the root of the interactive display function VIEW and the family of editors: EX, EDIT, VI.

- # Find lines by number or pattern.
- # Add, delete, change, copy, move or join lines.
- # Permute or split contents of a line.
- # Replace one or all instances of a pattern within a line.
- # Combine or split files.
- # Switch to the location of a 'tag'.
- # Enter intraline editing.
- # Reverse the effects of the last command.
- # Escape to Shell during editing.
- # Indent automatically.
- # Define abbreviations.
- # Attempt to recover the buffer in case of hangups or crashes.
- # Read and execute commands from a specified file.

- # Simulate an intelligent terminal on a dumb terminal.

EDIT — A small version of EX. Avoids some of the complexities of EX to provide an environment for new and casual users.

- # Find lines by number or pattern.
- # Add, delete, change, copy or move lines.
- # Replace a pattern in a line.
- # Add the contents of a file.
- # Reverse the effects of the last command.
- # Escape to Shell.
- # Attempt to recover the buffer in case of hangups or crashes.

VI — The screen oriented editor VI is based on EX (see above). Additional attributes:

- # Numerous commands for file manipulation.
  i.e. edit file containing the tag 'tag'
  at the first line of 'tag'
- # Extensive command set for scrolling, paging and cursor motion.
  i.e. move to the end of line
  move to the begin of the next word
- # Various units of text can be handled: words, sentences, sections.
  i.e. duplicate sentence
  delete word
- # Searching for strings by a set of different conditions.
  i.e. matches any character between 'x' and 'y'
  matches the end of a word
- # Definition of macros for saving time by typing commands.
- # Escape to the line oriented editor EX.

VIEW — Interactive display function. Works like the VI - but with read-only files.

CTAGS — Make a tags file for EX from the specified C, PASCAL and FORTRAN sources. A tags file gives the locations of specified objects (in this case functions) in a group of files.

MKSTR — Used to create a file of error messages by massaging C source code.

- # Places all error messages from a C source file in a specified file.
- # Keys on the string 'error(" to process the error messages to the message file.
- # The copy of the C source file contains pointer into the message file to retrieve the error message.

XSTR — Extract strings from C programs to implement shared constant strings.

- # Maintains a file into which strings of component parts of a large program are hashed.
- # The strings are replaced with references to the common area.

DICTION — Find wordy sentences in a document.

- # Finds all sentences that contain phrases from a data base of bad or wordy diction.
- # The user may supply his own pattern file.

EXPLAIN Interactive thesaurus for the phrases found by diction.

STYLE Analyze surface characteristics of a document.

- # Reports on
  - readability
  - sentence length and structure
  - word length and usage
  - verb type
  - sentence openers
- # Options to locate sentences with certain characteristics.

3.2. Document Formatting

VTROFF Troff for raster printer/plotter.

VFONTINFO Inspect and print out information about unix fonts.

SOELIM Eliminate .so's from nroff input.

FMT Simple text formatter.

- # Produces an output with lines as close to 72 characters as possible.
- # Spacing at the beginning of input lines and blank lines are preserved.

ME Technical paper layout package for use with NROFF, TROFF and VTROFF.

CHECKNR Check NROFF/TROFF files.

- # Knows about MS and ME macro packages.
- # Checks unknown commands.
- # Checks mismatched opening and closing delimiters in case of
  - macros which always come in pairs
  - font changes
  - size changes

PTI Interpret a stream from the standard output of TROFF as it would act on the typesetter.

4. Games

FISH Childrens' card guessing game.

HANGMAN

HANG Word guessing games. Uses the dictionary supplied with SPELL.

TWINKLE1

TWINKLE2 Milky way on the screen.

WORM Lead the worm to random points.

WORMS Several worms running around on screen.

Writing Papers with Nroff

Using – ME

Trademarks:

MUNIX, CADMUS	for PCS
DEC, PDP	for DEC
UNIX	for Bell Laboratories

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 .b 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.[1]

It is important that the footnote appears *inside* the quote, so that you can be sure that the footnote will appear on the same page as the quote.

4.2. Delayed Text

Delayed text is very similar to a footnote except that it is printed when called for explicitly. This allows a list of references to appear (for example) at the end of each chapter, as is the convention in some disciplines. Use *# on delayed text instead of ** as on footnotes.

If you are using delayed text as your standard reference mechanism, you can still use footnotes, except that you may want to reference them with special characters* rather than numbers.

4.3. Indexes

An "index" (actually more like a table of contents, since the entries are not sorted alphabetically) resembles delayed text, in that it is saved until called for. However, each entry has the page number (or some other tag) appended to the last line of the index entry after a row of dots.

Index entries begin with the request .(x and end with .)x. The .)x request may have a argument, which is the value to print as the "page number". It defaults to the current page number. If the page number given is an underscore ("_") no page number or line of dots is printed at all. To get the line of dots without a page number, type .)x "", which specifies an explicitly null page number.

The .xp request prints the index.

For example, the input:

---

[1] James R. Ware, *The Best of Confucius*, Halcyon House, 1950. Page 77.

*Such as an asterisk.

```
.(x
Sealing wax
.)x
.(x
Cabbages and kings
.)x _
.(x
Why the sea is boiling hot
.)x 2.5a
.(x
Whether pigs have wings
.)x "
.(x
This is a terribly long index entry, such as might be used
for a list of illustrations, tables, or figures; I expect it to
take at least two lines.
.)x
.xp
```

generates:

Sealing wax 9

Cabbages and kings

Why the sea is boiling hot 2.5a

Whether pigs have wings .....

This is a terribly long index entry, such as might be used for a list of illustrations, tables, or figures; I expect it to take at least two lines. 9

The .(x request may have a single character argument, specifying the "name" of the index; the normal index is x. Thus, several "indicies" may be maintained simultaneously (such as a list of tables, table of contents, etc.).

Notice that the index must be printed at the *end* of the paper, rather than at the beginning where it will probably appear (as a table of contents); the pages may have to be physically rearranged after printing.

5. Fancier Features

A large number of fancier requests exist, notably requests to provide other sorts of paragraphs, numbered sections of the form 1.2.3 (such as used in this document), and multicolumn output.

5.1. More Paragraphs

Paragraphs generally start with a blank line and with the first line indented. It is possible to get left-justified block-style paragraphs by using .lp instead of .pp, as demonstrated by the next paragraph.

Sometimes you want to use paragraphs that have the *body* indented, and the first line exdented (opposite of indented) with a label. This can be done with the .ip request. A word specified on the same line as .ip is printed in the margin, and the body is lined up at a prespecified position (normally five spaces). For example, the input:

```
.ip one
This is the first paragraph.
Notice how the first line
of the resulting paragraph lines up
with the other lines in the paragraph.
.ip two
And here we are at the second paragraph already.
You may notice that the argument to .ip
appears
in the margin.
.lp
We can continue text...
```

produces as output:

one This is the first paragraph. Notice how the first line of the resulting paragraph lines up with the other lines in the paragraph.

two And here we are at the second paragraph already. You may notice that the argument to .ip appears in the margin.

We can continue text without starting a new indented paragraph by using the .lp request.

If you have spaces in the label of a .ip request, you must use an "unpaddable space" instead of a regular space. This is typed as a backslash character ("\") followed by a space. For example, to print the label "Part 1", enter:

```
.ip "Part\ 1"
```

If a label of an indented paragraph (that is, the argument to .ip) is longer than the space allocated for the label, .ip will begin a new line after the label. For example, the input:

```
.ip longlabel
This paragraph had a long label.
The first character of text on the first line
will not line up with the text on second and subsequent lines,
although they will line up with each other.
```

will produce:

longlabel
This paragraph had a long label. The first character of text on the first line will not line up with the text on second and subsequent lines, although they will line up with each other.

It is possible to change the size of the label by using a second argument which is the size of the label. For example, the above example could be done correctly by saying:

```
.ip longlabel 10
```

which will make the paragraph indent 10 spaces for this paragraph only. If you have many paragraphs to indent all the same amount, use the *number register* ii. For example, to leave one inch of space for the label, type:

```
.nr ii 1i
```

somewhere before the first call to .ip. Refer to the reference manual for more information.

If .ip is used with no argument at all no hanging tag will be printed. For example, the input:

```
.ip [a]
This is the first paragraph of the example.
We have seen this sort of example before.
.ip
This paragraph is lined up with the previous paragraph,
but it has no tag in the margin.
```

produces as output:

[a] This is the first paragraph of the example. We have seen this sort of example before.

This paragraph is lined up with the previous paragraph, but it has no tag in the margin.

A special case of .lp 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 these processors for a description of their use.

The eqn and neqn programs are described fully in the document *Typesetting Mathematics − Users' Guide* by Brian W. Kernighan and Lorinda L. Cherry. Equations are centered, and are kept on one page. They are introduced by the .EQ request and terminated by the .EN request.

The .EQ request may take an equation number as an optional argument, which is printed vertically centered on the right hand side of the equation. If the equation becomes too long it should be split between two lines. To do this, type:

```
.EQ (eq 34)
text of equation 34
.EN C
.EQ
continuation of equation 34
.EN
```

The C on the .EN request specifies that the equation will be continued.

The tbl program produces tables. It is fully described (including numerous examples) in the document *Tbl − A Program to Format Tables* by M. E. Lesk. Tables begin with the .TS request and end with the .TE request. Tables are normally kept on a single page. If you have a table which is too big to fit on a single page, so that you know it will extend to several pages, begin the table with the request .TS H and put the request .TH

```
.th                             \" set for thesis mode
.fo "DRAFT"                     \" define footer for each page
.tp                             \" begin title page
.(l C                           \" center a large block
THE GROWTH OF TOENAILS
IN UPPER PRIMATES
.sp
by
.sp
Frank Furter
.)l                             \" end centered part
.+c INTRODUCTION                \" begin chapter named "INTRODUCTION"
.(x t                           \" make an entry into index 't'
Introduction
.)x                             \" end of index entry
text of chapter one
.+c "NEXT CHAPTER"              \" begin another chapter
.(x t                           \" enter into index 't' again
Next Chapter
.)x
text of chapter two
.+c CONCLUSIONS
.(x t
Conclusions
.)x
text of chapter three
.++ B                           \" begin bibliographic information
.+c BIBLIOGRAPHY                \" begin another 'chapter'
.(x t
Bibliography
.)x
text of bibliography
.++ P                           \" begin preliminary material
.+c "TABLE OF CONTENTS"
.xp t                           \" print index 't' collected above
.+c PREFACE                     \" begin another preliminary section
text of preface
```

Figure 2. Outline of a Sample Paper

after the part of the table which you want duplicated at the top of every page that the table is printed on. For example, a table definition for a long table might look like:

```
.TS H
c s s
n n n.
THE TABLE TITLE
.TH
text of the table
.TE
```

5.5. Two Column Output

You can get two column output automatically by using the request .2c. This causes everything after it to be output in two-column form. The request .bc will start a new column; it differs from .bp in that .bp may leave a totally blank column when it starts a new page. To revert to single column output, use .1c.

5.6. Defining Macros

A *macro* is a collection of requests and text which may be used by stating a simple request. Macros begin with the line .de *xx* (where *xx* is the name of the macro to be defined) and end with the line consisting of two dots. After defining the macro, stating the line .*xx* is the same as stating all the other lines. For example, to define a macro that spaces 3 lines and then centers the next input line, enter:

```
.de SS
.sp 3
.ce
```

and use it by typing:

```
.SS
Title Line
(beginning of text)
```

Macro names may be one or two characters. In order to avoid conflicts with names in −me, always use upper case letters as names. The only names to avoid are TS, TH, TE, EQ, and EN.

5.7. Annotations Inside Keeps

Sometimes you may want to put a footnote or index entry inside a keep. For example, if you want to maintain a "list of figures" you will want to do something like:

```
.(z
.(c
text of figure
.)c
.ce
Figure 5.
.(x f
Figure 5
.)x
.)z
```

which you may hope will give you a figure with a label and an entry in the index *f* (presumably a list of figures index). Unfortunately, the index entry is read and interpreted when the keep is read, not when it is printed, so the page number in the index is likely to be wrong. The solution is to use the magic string \! at the beginning of all the lines dealing with the index. In other words, you should use:

```
.(z
.(c
Text of figure
.)c
.ce
Figure 5.
\!.(x f
\!Figure 5
\!.)x
.)z
```

which will defer the processing of the index until the figure is output. This will guarantee that the page number in the index is correct. The same comments apply to blocks (with .(b and .)b) as well.

6. TROFF and the Photosetter

With a little care, you can prepare documents that will print nicely on either a regular terminal or when phototypeset using the TROFF formatting program.

6.1. Fonts

A *font* is a style of type. There are three fonts that are available simultaneously, Times Roman, Times Italic, and Times Bold, plus the special math font. The normal font is Roman. Text which would be underlined in NROFF with the .ul request is set in italics in TROFF.

There are ways of switching between fonts. The requests .r, .I, and .b switch to Roman, italic, and bold fonts respectively. You can set a single word in some font by typing (for example):

```
.i word
```

which will set *word* in italics but does not affect the surrounding text. In NROFF, italic and bold text is underlined.

Notice that if you are setting more than one word in whatever font, you must surround that word with double quote marks (`""`) so that it will appear to the NROFF processor as a single word. The quote marks will not appear in the formatted text. If you do want a quote mark to appear, you should quote the entire string (even if a single word), and use *two* quote marks where you want one to appear. For example, if you want to produce the text:

"*Master Control*"

in italics, you must type:

```
.i ""Master Control\""
```

The \| produces a very narrow space so that the "I" does not overlap the quote sign in TROFF, like this:

"*Master Control*"

There are also several "pseudo-fonts" available. The input:

```
.(b
.u underlined
.bi "bold italics"
.bx "words in a box"
.)b
```

generates

underlined
bold italics
words in a box

In NROFF these all just underline the text. Notice that pseudo font requests set only the single parameter in the pseudo font; ordinary font requests will begin setting all text in the special font if you do not provide a parameter. No more than one word should appear with these three font requests in the middle of lines. This is because of the way TROFF justifies text. For example, if you were to issue the requests:

.bi "some bold italics"
and
.bx "words in a box"

in the middle of a line TROFF would produce some bold italics and |words in a box|, which I think you will agree does not look good.

The second parameter of all font requests is set in the original font. For example, the font request:

.b bold face

generates "bold" in bold font, but sets "face" in the font of the surrounding text, resulting in:

boldface.

To set the two words bold and face both in bold face, type:

.b "bold face"

You can mix fonts in a word by using the special sequence \c at the end of a line to indicate "continue text processing"; this allows input lines to be joined together without a space inbetween them. For example, the input:

.u under \c
.i italics

generates under*italics*, but if we had typed:

.u under
.i italics

the result would have been under *italics* as two words.

6.2. Point Sizes

The phototypesetter supports different sizes of type, measured in points. The default point size is 10 points for most text, 8 points for footnotes. To change the pointsize, type:

.sz +*N*

where *N* is the size wanted in points. The *vertical spacing* (distance between the bottom of most letters (the *baseline*) between adjacent lines) is set to be proportional to the type size.

Warning: changing point sizes on the phototypesetter is a slow mechanical operation. Size changes should be considered carefully.

6.3. Quotes

It is conventional when using the typesetter to use pairs of grave and acute accents to generate double quotes, rather than the double quote character (`"`). This is because it looks better to use grave and acute accents; for example, compare "quote" to ``quote''

In order to make quotes compatible between the typesetter and terminals, you may use the sequences *(lq and *(rq to stand for the left and right quote respectively.

These both appear as " on most terminals, but are typeset as `` and '' respectively. For example, use:

```
\*(lqSome things aren't true
even if they did happen.\*(rq
```

to generate the result:

``Some things aren't true even if they did happen.''

As a shorthand, the special font request:

```
.q "quoted text"
```

will generate ``quoted text''. Notice that you must surround the material to be quoted with double quote marks if it is more than one word.

Acknowledgments

I would like to thank Bob Epstein, Bill Joy, and Larry Rowe for having the courage to use the −me macros to produce non-trivial papers during the development stages; Ricki Blau, Pamela Humphrey, and Jim Joyce for their help with the documentation phase; and the plethora of people who have contributed ideas and have given support for the project.

This document was TROFF'ed on December 18, 1979 and applies to version 1.1 of the −me macros.

−ME REFERENCE MANUAL

Release 1.1/25

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 December 14, 1979 and applies to version 1.1/25 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: .se, .lo, .th, or .ae. Also, the effects of changing parameters which will have a global effect on the format of the page (notably page length and header and footer margins) are not well defined and should be avoided.

.lp — Begin left-justified paragraph. Centering and underlining are turned off if they were on, the font is set to \n(pf [1] the type size is set to \n(pp [10p], and a \n(ps space is inserted before the paragraph [0.35v in TROFF, 1v or 0.5v in NROFF depending on device resolution]. The indent is reset to \n(SI [0] plus \n(po [0] unless the paragraph is inside a display. (see .ba). At least the first two lines of the paragraph are kept together on a page.

.pp — Like .lp, except that it puts \n(pi [5n] units of indent. This is the standard paragraph macro.

.lp *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. If *T* will not fit in the space provided, .lp will start a new line.

.np — A variant of .lp 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 [B] 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

.lx +*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.

.ll +*N* — Set the line length to *N* [6.0i]. This differs from .ll because it only affects the current environment.

.ll +*N* — Set line length in all environments to *N* [6.0i]. This should not be used after output has begun, and particularly not in two-columned output. The current line length is stored in \n(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. The footnote numbering is reset to one.

.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 ('['). Extra space is left above the line to allow room for the superscript.

*] — Unsuperscript. Inverse to *[. For example, to produce a superscript you might type x*[2*], which will produce x^2.

*< — Subscript. Defaults to '<' if half-carriage motion not possible. Extra space is left below the line to allow for the subscript.

*> — Inverse to *<.

*(dw	The day of the week, as a word.
*(mo	The month, as a word.
*(td	Today's date, directly printable. The date is of the form December 14, 1979. Other forms of the date can be used by using \n(dy (the day of the month; for example, 14), *(mo (as noted above) or \n(mo (the same, but as an ordinal number; for example, December is 12), and \n(yr (the last two digits of the current year).
*(lq	Left quote marks. Double quote in NROFF.
*(rq	Right quote.
*−	¼ em dash in TROFF; two hyphens in NROFF.

13. Special Characters and Marks

There are a number of special characters and diacritical marks (such as accents) available through −me. To reference these characters, you must call the macro .sc to define the characters before using them.

.sc	Define special characters and diacritical marks, as described in the remainder of this section. This macro must be stated before initialization.

The special characters available are listed below.

Name	Usage	Example	
Acute accent	*'	a*'	á
Grave accent	*`	e*`	è
Umlat	*:	u*:	ü
Tilde	*~	n*~	ñ
Caret	*^	e*^	ê
Cedilla	*,	c*,	ç
Czech	*v	e*v	ě
Circle	*o	A*o	Å
There exists	*(qe		∃
For all	*(qa		∀

Acknowledgments

I would like to thank Bob Epstein, Bill Joy, and Larry Rowe for having the courage to use the −me macros to produce non-trivial papers during the development stages; Ricki Blau, Pamela Humphrey, and Jim Joyce for their help with the documentation phase; and the plethora of people who have contributed ideas and have given support for the project.

Writing Tools

The Style and Diction Programs

Text processing systems are now in heavy use in many companies to format documents. With many documents stored on line, it has become possible to use computers to study writing style itself and to help writers produce better written and more readable prose. The system of programs described here is an initial step toward such help. It includes programs and data base designed to produce a stylistic profile of writing at the word and sentence level. The system measures readability, sentence and word lenght, sentence type, word usage, and sentence openers. It also locates common examples of wordy phrasing and bad diction. The system is useful for evaluating a document's style, locating sentences that may be difficult to read or excessively wordy, and determining a particular writer's style over several documents.

Writing Tools - The STYLE and DICTION Programs

L. L. Cherry

Bell Laboratories
Murray Hill, New Jersey 07974

W. Vesterman

Livingston College
Rutgers University

1. Introduction

Computers have become important in the document preparation process, with programs to check for spelling errors and to format documents. As the amount of text stored on line increases, it becomes feasible and attractive to study writing style and to attempt to help the writer in producing readable documents. The system of writing tools described here is a first step toward such help. The system includes programs and a data base to analyze writing style at the word and sentence level. We use the term "style" in this paper to describe the results of a writer's particular choices among individual words and sentence forms. Although many judgements of style are subjective, particularly those of word choice, there are some objective measures that experts agree lead to good style. Three programs have been written to measure some of the objectively definable characteristics of writing style and to identify some commonly misused or unnecessary phrases. Although a document that conforms to the stylistic rules is not guaranteed to be coherent and readable, one that violates all of the rules is likely to be difficult or tedious to read. The program STYLE calculates readability, sentence length variability, sentence type, word usage and sentence openers at a rate of about 400 words per second on a PDP11/70 running the UNIX† Operating System. It assumes that the sentences are well-formed, i. e. that each sentence has a verb and that the subject and verb agree in number. DICTION identifies phrases that are either bad usage or unnecessarily wordy. EXPLAIN acts as a thesaurus for the phrases found by DICTION. Sections 2, 3, and 4 describe the programs; Section 5 gives the results on a cross-section of technical documents; Section 6 discusses accuracy and problems; Section 7 gives implementation details.

2. STYLE

The program STYLE reads a document and prints a summary of readability indices, sentence length and type, word usage, and sentence openers. It may also be used to locate all sentences in a document longer than a given length, of readability index higher than a given number, those containing a passive verb, or those beginning with an expletive. STYLE is based on the system for finding English word classes or parts of speech, PARTS [1]. PARTS is a set of programs that uses a small dictionary (about 350 words) and suffix rules to partially assign word classes to English text. It then uses experimentally derived rules of word order to assign word classes to all words in the text with an accuracy of about 95%. Because PARTS uses only a small dictionary and general rules, it works on text about any subject, from physics to psychology. Style measures have been built into the output phase of the programs that make up PARTS. Some of the measures are simple counters of the word classes found by PARTS; many are more complicated. For example, the verb count is the total number of verb phrases. This includes phrases like:

---

†UNIX is a Trademark of Bell Laboratories.

has been going
was only going
to go

each of which each counts as one verb. Figure 1 shows the output of STYLE run on a paper by Kernighan and Mashey about the UNIX programming environment [2].

```
programming environment
readability grades:
                        (Kincaid) 12.3  (auto) 12.8  (Coleman-Liau) 11.8  (Flesch) 13.5 (46.3)
sentence info:
                        no. sent 335 no. wds 7419
                        av sent leng 22.1 av word leng 4.91
                        no. questions 0 no. imperatives 0
                        no. nonfunc wds 4362  58.8%   av leng 6.38
                        short sent (<17) 35% (118) long sent (>32)  16% (55)
                        longest sent 82 wds at sent 174; shortest sent 1 wds at sent 117
sentence types:
                        simple  34% (114) complex  32% (108)
                        compound  12% (41) compound-complex  21% (72)
word usage:
                        verb types as % of total verbs
                        tobe  45% (373) aux  16% (133) inf  14% (114)
                        passives as % of non-inf verbs  20% (144)
                        types as % of total
                        prep 10.8% (804) conj 3.5% (262) adv 4.8% (354)
                        noun 26.7% (1983) adj 18.7% (1388) pron 5.3% (393)
                        nominalizations  2 % (155)
sentence beginnings:
                        subject opener: noun (63) pron (43) pos (0) adj (58) art (62) tot  67%
                        prep  12% (39) adv  9% (31)
                        verb  0% (1)  sub_conj  6% (20) conj  1% (5)
                        expletives  4% (13)
```

Figure 1

As the example shows, STYLE output is in five parts. After a brief discussion of sentences, we will describe the parts in order.

2.1. What is a sentence?

Readers of documents have little trouble deciding where the sentences end. People don't even have to stop and think about uses of the character "." in constructions like 1.25, A. J. Jones, Ph.D., i. e., or etc. When a computer reads a document, finding the end of sentences is not as easy. First we must throw away the printer's marks and formatting commands that litter the text in computer form. Then STYLE defines a sentence as a string of words ending in one of:

! ? /.

The end marker "/." may be used to indicate an imperative sentence. Imperative sentences that are not so marked are not identified as imperative. STYLE properly handles numbers with embedded decimal points and commas, strings of letters and numbers with embedded decimal points used for naming computer file names, and the common abbreviations listed in Appendix

1. Numbers that end sentences, like the preceding sentence, cause a sentence break if the next word begins with a capital letter. Initials only cause a sentence break if the next word begins with a capital and is found in the dictionary of function words used by PARTS. So the string

J. D. JONES

does not cause a break, but the string

– ... system H. The ...

does. With these rules most sentences are broken at the proper place, although occasionally either two sentences are called one or a fragment is called a sentence. More on this later.

2.2. Readability Grades

The first section of STYLE output consists of four readability indices. As Klare points out in [3] readability indices may be used to estimate the reading skills needed by the reader to understand a document. The readability indices reported by STYLE are based on measures of sentence and word lengths. Although the indices may not measure whether the document is coherent and well organized, experience has shown that high indices seem to be indicators of stylistic difficulty. Documents with short sentences and short words have low scores; those with long sentences and many polysyllabic words have high scores. The 4 formulae reported are Kincaid Formula [4], Automated Readability Index [5], Coleman-Liau Formula [6] and a normalized version of Flesch Reading Ease Score [7]. The formulae differ because they were experimentally derived using different texts and subject groups. We will discuss each of the formulae briefly; for a more detailed discussion the reader should see [3].

The Kincaid Formula, given by:

$$Reading\_Grade = 11.8 * syl\_per\_wd + .39 * wds\_per\_sent - 15.59$$

was based on Navy training manuals that ranged in difficulty from 5.5 to 16.3 in reading grade level. The score reported by this formula tends to be in the mid-range of the 4 scores. Because it is based on adult training manuals rather than school book text, this formula is probably the best one to apply to technical documents.

The Automated Readability Index (ARI), based on text from grades 0 to 7, was derived to be easy to automate. The formula is:

$$Reading\_Grade = 4.71 * let\_per\_wd + .5 * wds\_per\_sent - 21.43$$

ARI tends to produce scores that are higher than Kincaid and Coleman-Liau but are usually slightly lower than Flesch.

The Coleman-Liau Formula, based on text ranging in difficulty from .4 to 16.3, is:

$$Reading\_Grade = 5.89 * let\_per\_wd - .3 * sent\_per\_100\_wds - 15.8$$

Of the four formulae this one usually gives the lowest grade when applied to technical documents.

The last formula, the Flesch Reading Ease Score, is based on grade school text covering grades 3 to 12. The formula, given by:

$$Reading\_Score = 206.835 - 84.6 * syl\_per\_wd - 1.015 * wds\_per\_sent$$

is usually reported in the range 0 (very difficult) to 100 (very easy). The score reported by STYLE is scaled to be comparable to the other formulas, except that the maximum grade level reported is set to 17. The Flesch score is usually the highest of the 4 scores on technical documents.

Coke [8] found that the Kincaid Formula is probably the best predictor for technical documents; both ARI and Flesch tend to overestimate the difficulty; Coleman-Liau tend to underestimate. On text in the range of grades 7 to 9 the four formulas tend to be about the same. On easy text the Coleman-Liau formula is probably preferred since it is reasonably accurate at the

lower grades and it is safer to present text that is a little too easy than a little too hard.

If a document has particularly difficult technical content, especially if it includes a lot of mathematics, it is probably best to make the text very easy to read, i.e. a lower readability index by shortening the sentences and words. This will allow the reader to concentrate on the technical content and not the long sentences. The user should remember that these indices are estimators; they should not be taken as absolute numbers. STYLE called with "−r number" will print all sentences with an Automated Readability Index equal to or greater than "number".

2.3. Sentence length and structure

The next two sections of STYLE output deal with sentence length and structure. Almost all books on writing style or effective writing emphasize the importance of variety in sentence length and structure for good writing. Ewing's first rule in discussing style in the book *Writing for Results* [9] is:

> "Vary the sentence structure and length of your sentences."

Leggett, Mead and Charvat break this rule into 3 in *Prentice-Hall Handbook for Writers* [10] as follows:

> "34a. Avoid the overuse of short simple sentences."
> "34b. Avoid the overuse of long compound sentences."
> "34c. Use various sentence structures to avoid monotony and increase effectiveness."

Although experts agree that these rules are important, not all writers follow them. Sample technical documents have been found with almost no sentence length or type variability. One document had 90% of its sentences about the same length as the average; another was made up almost entirely of simple sentences (80%).

The output sections labeled "sentence info" and "sentence types" give both length and structure measures. STYLE reports on the number and average length of both sentences and words, and number of questions and imperative sentences (those ending in "/."). The measures of non-function words are an attempt to look at the content words in the document. In English non-function words are nouns, adjectives, adverbs, and non-auxiliary verbs; function words are prepositions, conjunctions, articles, and auxiliary verbs. Since most function words are short, they tend to lower the average word length. The average length of non-function words may be a more useful measure for comparing word choice of different writers than the total average word length. The percentages of short and long sentences measure sentence length variability. Short sentences are those at least 5 words less than the average; long sentences are those at least 10 words longer than the average. Last in the sentence information section is the length and location of the longest and shortest sentences. If the flag "−l number" is used, STYLE will print all sentences longer than "number".

Because of the difficulties in dealing with the many uses of commas and conjunctions in English, sentence type definitions vary slightly from those of standard textbooks, but still measure the same constructional activity.

1. A simple sentence has one verb and no dependent clause.
2. A complex sentence has one independent clause and one dependent clause, each with one verb. Complex sentences are found by identifying sentences that contain either a subordinate conjunction or a clause beginning with words like "that" or "who". The preceding sentence has such a clause.
3. A compound sentence has more than one verb and no dependent clause. Sentences joined by ";" are also counted as compound.
4. A compound-complex sentence has either several dependent clauses or one dependent clause and a compound verb in either the dependent or independent clause.

Even using these broader definitions, simple sentences dominate many of the technical documents that have been tested, but the example in Figure 1 shows variety in both sentence

structure and sentence length.

2.4. Word Usage

The word usage measures are an attempt to identify some other constructional features of writing style. There are many different ways in English to say the same thing. The constructions differ from one another in the form of the words used. The following sentences all convey approximately the same meaning but differ in word usage:

> The cxio program is used to perform all communication between the systems.
> The cxio program performs all communications between the systems.
> The cxio program is used to communicate between the systems.
> The cxio program communicates between the systems.
> All communication between the systems is performed by the cxio program.

The distribution of the parts of speech and verb constructions helps identify overuse of particular constructions. Although the measures used by STYLE are crude, they do point out problem areas. For each category, STYLE reports a percentage and a raw count. In addition to looking at the percentage, the user may find it useful to compare the raw count with the number of sentences. If, for example, the number of infinitives is almost equal to the number of sentences, then many of the sentences in the document are constructed like the first and third in the preceding example. The user may want to transform some of these sentences into another form. Some of the implications of the word usage measures are discussed below.

Verbs are measured in several different ways to try to determine what types of verb constructions are most frequent in the document. Technical writing tends to contain many passive verb constructions and other usage of the verb "to be". The category of verbs labeled "tobe" measures both passives and sentences of the form:

> *subject tobe predicate*

In counting verbs, whole verb phrases are counted as one verb. Verb phrases containing auxiliary verbs are counted in the category "aux". The verb phrases counted here are those whose tense is not simple present or simple past. It might eventually be useful to do more detailed measures of verb tense or mood. Infinitives are listed as "inf". The percentages reported for these three categories are based on the total number of verb phrases found. These categories are not mutually exclusive; they cannot be added, since, for example, "to be going" counts as both "tobe" and "inf". Use of these three types of verb constructions varies significantly among authors.

STYLE reports passive verbs as a percentage of the finite verbs in the document. Most style books warn against the overuse of passive verbs. Coleman [11] has shown that sentences with active verbs are easier to learn than those with passive verbs. Although the inverted object-subject order of the passive voice seems to emphasize the object, Coleman's experiments showed that there is little difference in retention by word position. He also showed that the direct object of an active verb is retained better than the subject of a passive verb. These experiments support the advice of the style books suggesting that writers should try to use active verbs wherever possible. The flag "−p" causes STYLE to print all sentences containing passive verbs.

Pronouns add cohesiveness and connectivity to a document by providing back-reference. They are often a short-hand notation for something previously mentioned, and therefore connect the sentence containing the pronoun with the word to which the pronoun refers. Although there are other mechanisms for such connections, documents with no pronouns tend to be wordy and to have little connectivity.

Adverbs can provide transition between sentences and order in time and space. In performing these functions, adverbs, like pronouns, provide connectivity and cohesiveness.

Conjunctions provide parallelism in a document by connecting two or more equal units. These units may be whole sentences, verb phrases, nouns, adjectives, or prepositional phrases. The compound and compound-complex sentences reported under sentence type are parallel structures. Other uses of parallel structures are indicated by the degree that the number of conjunctions reported under word usage exceeds the compound sentence measures.

Nouns and Adjectives. A ratio of nouns to adjectives near unity may indicate the over-use of modifiers. Some technical writers qualify every noun with one or more adjectives. Qualifiers in phrases like "simple linear single-link network model" often lend more obscurity than precision to a text.

Nominalizations are verbs that are changed to nouns by adding one of the suffixes "ment", "ance", "ence", or "ion". Examples are accomplishment, admittance, adherence, and abbreviation. When a writer transforms a nominalized sentence to a non-nominalized sentence, she/he increases the effectiveness of the sentence in several ways. The noun becomes an active verb and frequently one complicated clause becomes two shorter clauses. For example,

> Their inclusion of this provision is admission of the importance of the system.
> When they included this provision, they admitted the importance of the system.

Coleman found that the transformed sentences were easier to learn, even when the transformation produced sentences that were slightly longer, provided the transformation broke one clause into two. Writers who find their document contains many nominalizations may want to transform some of the sentences to use active verbs.

2.5. Sentence openers

Another agreed upon principle of style is variety in sentence openers. Because STYLE determines the type of sentence opener by looking at the part of speech of the first word in the sentence, the sentences counted under the heading "subject opener" may not all really begin with the subject. However, a large percentage of sentences in this category still indicates lack of variety in sentence openers. Other sentence opener measures help the user determine if there are transitions between sentences and where the subordination occurs. Adverbs and conjunctions at the beginning of sentences are mechanisms for transition between sentences. A pronoun at the beginning shows a link to something previously mentioned and indicates connectivity.

The location of subordination can be determined by comparing the number of sentences that begin with a subordinator with the number of sentences with complex clauses. If few sentences start with subordinate conjunctions then the subordination is embedded or at the end of the complex sentences. For variety the writer may want to transform some sentences to have leading subordination.

The last category of openers, expletives, is commonly overworked in technical writing. Expletives are the words "it" and "there", usually with the verb "to be", in constructions where the subject follows the verb. For example,

> There are three streets used by the traffic.
> There are too many users on this system.

This construction tends to emphasize the object rather than the subject of the sentence. The flag "−e" will cause STYLE to print all sentences that begin with an expletive.

3. DICTION

The program DICTION prints all sentences in a document containing phrases that are either frequently misused or indicate wordiness. The program, an extension of Aho's FGREP [12] string matching program, takes as input a file of phrases or patterns to be matched and a file of text to be searched. A data base of about 450 phrases has been compiled as a default pattern file for DICTION. Before attempting to locate phrases, the program maps upper case letters to lower case and substitutes blanks for punctuation. Sentence boundaries were deemed less critical in DICTION than in STYLE, so abbreviations and other uses of the character "." are not treated specially. DICTION brackets all pattern matches in a sentence with the characters "[" "]" Although many of the phrases in the default data base are correct in some contexts, in others they indicate wordiness. Some examples of the phrases and suggested alternatives are:

Phrase	Alternative
a large number of	many
arrive at a decision	decide
collect together	collect
for this reason	so
pertaining to	about
through the use of	by or with
utilize	use
with the exception of	except

Appendix 2 contains a complete list of the default file. Some of the entries are short forms of problem phrases. For example, the phrase "the fact" is found in all of the following and is sufficient to point out the wordiness to the user:

Phrase	Alternative
accounted for by the fact that	caused by
an example of this is the fact that	thus
based on the fact that	because
despite the fact that	although
due to the fact that	because
in light of the fact that	because
in view of the fact that	since
notwithstanding the fact that	although

Entries in Appendix 2 preceded by "~" are not matched. See Section 7 for details on the use of "~".

The user may supply her/his own pattern file with the flag "-f patfile". In this case the default file will be loaded first, followed by the user file. This mechanism allows users to suppress patterns contained in the default file or to include their own pet peeves that are not in the default file. The flag "-n" will exclude the default file altogether. In constructing a pattern file, blanks should be used before and after each phrase to avoid matching substrings in words. For example, to find all occurrences of the word "the", the pattern " the " should be used. The blanks cause only the word "the" to be matched and not the string "the" in words like there, other, and therefore. One side effect of surrounding the words with blanks is that when two phrases occur without intervening words, only the first will be matched.

4. EXPLAIN

The last program, EXPLAIN, is an interactive thesaurus for phrases found by DICTION. The user types one of the phrases bracketed by DICTION and EXPLAIN responds with suggested substitutions for the phrase that will improve the diction of the document.

Table 1
Text Statistics on 20 Technical Documents

	variable	minimum	maximum	mean	standard deviation
Readability	Kincaid	9.5	16.9	13.3	2.2
	automated	9.0	17.4	13.3	2.5
	Cole-Liau	10.0	16.0	12.7	1.8
	Flesch	8.9	17.0	14.4	2.2
sentence info.	av sent length	15.5	30.3	21.6	4.0
	av word length	4.61	5.63	5.08	.29
	av nonfunction length	5.72	7.30	6.52	.45
	short sent	23%	46%	33%	5.9
	long sent	7%	20%	14%	2.9
sentence types	simple	31%	71%	49%	11.4
	complex	19%	50%	33%	8.3
	compound	2%	14%	7%	3.3
	compound-complex	2%	19%	10%	4.8
verb types	tobe	26%	64%	44.7%	10.3
	auxiliary	10%	40%	21%	8.7
	infinitives	8%	24%	15.1%	4.8
	passives	12%	50%	29%	9.3
word usage	prepositions	10.1%	15.0%	12.3%	1.6
	conjunction	1.8%	4.8%	3.4%	.9
	adverbs	1.2%	5.0%	3.4%	1.0
	nouns	23.6%	31.6%	27.8%	1.7
	adjectives	15.4%	27.1%	21.1%	3.4
	pronouns	1.2%	8.4%	2.5%	1.1
	nominalizations	2%	5%	3.3%	.8
sentence openers	prepositions	6%	19%	12%	3.4
	adverbs	0%	20%	9%	4.6
	subject	56%	85%	70%	8.0
	verbs	0%	4%	1%	1.0
	subordinating conj	1%	12%	5%	2.7
	conjunctions	0%	4%	0%	1.5
	expletives	0%	6%	2%	1.7

5. Results

5.1. STYLE

To get baseline statistics and check the program's accuracy, we ran STYLE on 20 technical documents. There were a total of 3287 sentences in the sample. The shortest document was 67 sentences long; the longest 339 sentences. The documents covered a wide range of subject matter, including theoretical computing, physics, psychology, engineering, and affirmative action. Table 1 gives the range, median, and standard deviation of the various style measures. As you will note most of the measurements have a fairly wide range of values across the sample documents.

As a comparison, Table 2 gives the median results for two different technical authors, a sample of instructional material, and a sample of the Federalist Papers. The two authors show similar styles, although author 2 uses somewhat shorter sentences and longer words than author 1. Author 1 uses all types of sentences, while author 2 prefers simple and complex sentences, using few compound or compound-complex sentences. The other major difference in the styles of these authors is the location of subordination. Author 1 seems to prefer embedded or trailing subordination, while author 2 begins many sentences with the subordinate clause. The

documents tested for both authors 1 and 2 were technical documents, written for a technical audience. The instructional documents, which are written for craftspeople, vary surprisingly little from the two technical samples. The sentences and words are a little longer, and they contain many passive and auxiliary verbs, few adverbs, and almost no pronouns. The instructional documents contain many imperative sentences, so there are many sentence with verb openers. The sample of Federalist Papers contrasts with the other samples in almost every way.

Table 2
Text Statistics on Single Authors

	variable	author 1	author 2	inst.	FED
readability	Kincaid	11.0	10.3	10.8	16.3
	automated	11.0	10.3	11.9	17.8
	Coleman-Liau	9.3	10.1	10.2	12.3
	Flesch	10.3	10.7	10.1	15.0
sentence info	av sent length	22.64	19.61	22.78	31.85
	av word length	4.47	4.66	4.65	4.95
	av nonfunction length	5.64	5.92	6.04	6.87
	short sent	35%	43%	35%	40%
	long sent	18%	15%	16%	21%
sentence types	simple	36%	43%	40%	31%
	complex	34%	41%	37%	34%
	compound	13%	7%	4%	10%
	compound-complex	16%	8%	14%	25%
verb type	tobe	42%	43%	45%	37%
	auxiliary	17%	19%	32%	32%
	infinitives	17%	15%	12%	21%
	passives	20%	19%	36%	20%
word usage	prepositions	10.0%	10.8%	12.3%	15.9%
	conjunctions	3.2%	2.4%	3.9%	3.4%
	adverbs	5.05%	4.6%	3.5%	3.7%
	nouns	27.7%	26.5%	29.1%	24.9%
	adjectives	17.0%	19.0%	15.4%	12.4%
	pronouns	5.3%	4.3%	2.1%	6.5%
	nominalizations	1%	2%	2%	3%
sentence openers	prepositions	11%	14%	6%	5%
	adverbs	9%	9%	6%	4%
	subject	65%	59%	54%	66%
	verb	3%	2%	14%	2%
	subordinating conj	8%	14%	11%	3%
	conjunction	1%	0%	0%	3%
	expletives	3%	3%	0%	3%

5.2. DICTION

In the few weeks that DICTION has been available to users about 35,000 sentences have been run with about 5,000 string matches. The authors using the program seem to make the suggested changes about 50-75% of the time. To date, almost 200 of the 450 strings in the default file have been matched. Although most of these phrases are valid and correct in some contexts, the 50-75% change rate seems to show that the phrases are used much more often than concise diction warrants.

6. Accuracy

6.1. Sentence Identification

The correctness of the STYLE output on the 20 document sample was checked in detail. STYLE misidentified 129 sentence fragments as sentences and incorrectly joined two or more sentences 75 times in the 3287 sentence sample. The problems were usually because of non-standard formatting commands, unknown abbreviations, or lists of non-sentences. An impossibly long sentence found as the longest sentence in the document usually is the result of a long list of non-sentences.

6.2. Sentence Types

Style correctly identified sentence type on 86.5% of the sentences in the sample. The type distribution of the sentences was 52.5% simple, 29.9% complex, 8.5% compound and 9% compound-complex. The program reported 49.5% simple, 31.9% complex, 8% compound and 10.4% compound-complex. Looking at the errors on the individual documents, the number of simple sentences was under-reported by about 4% and the complex and compound-complex were over-reported by 3% and 2%, respectively. The following matrix shows the programs output vs. the actual sentence type.

		Program Results			
		simple	complex	compound	comp-complex
Actual	simple	1566	132	49	17
Sentence	complex	47	892	6	65
Type	compound	40	6	207	23
	comp-complex	0	52	5	249

The system's inability to find imperative sentences seems to have little effect on most of the style statistics. A document with half of its sentences imperative was run, with and without the imperative end marker. The results were identical except for the expected errors of not finding verbs as sentence openers, not counting the imperative sentences, and a slight difference (1%) in the number of nouns and adjectives reported.

6.3. Word Usage

The accuracy of identifying word types reflects that of PARTS, which is about 95% correct. The largest source of confusion is between nouns and adjectives. The verb counts were checked on about 20 sentences from each document and found to be about 98% correct.

7. Technical Details

7.1. Finding Sentences

The formatting commands embedded in the text increase the difficulty of finding sentences. Not all text in a document is in sentence form: there are headings, tables, equations and lists, for example. Headings like "Finding Sentences" above should be discarded, not attached to the next sentence. However, since many of the documents are formatted to be phototypeset, and contain font changes, which usually operate on the most important words in the document, discarding all formatting commands is not correct. To improve the programs' ability to find sentence boundaries, the deformatting program, DEROFF [13], has been given some knowledge of the formatting packages used on the UNIX operating system. DEROFF will now do the following:

1. Suppress all formatting macros that are used for titles, headings, author's name, etc.

2. Suppress the arguments to the macros for titles, headings, author's name, etc.
3. Suppress displays, tables, footnotes and text that is centered or in no-fill mode.
4. Substitute a place holder for equations and check for hidden end markers. The place holder is necessary because many typists and authors use the equation setter to change fonts on important words. For this reason, header files containing the definition of the EQN delimiters must also be included as input to STYLE. End markers are often hidden when an equation ends a sentence and the period is typed inside the EQN delimiters.
5. Add a ". " after lists. If the flag −ml is also used, all lists are suppressed. This is a separate flag because of the variety of ways the list macros are used. Often, lists are sentences that should be included in the analysis. The user must determine how lists are used in the document to be analyzed.

Both STYLE and DICTION call DEROFF before they look at the text. The user should supply the −ml flag if the document contains many lists of non-sentences that should be skipped.

7.2. Details of DICTION

The program DICTION is based on the string matching program FGREP. FGREP takes as input a file of patterns to be matched and a file to be searched and outputs each line that contains any of the patterns with no indication of which pattern was matched. The following changes have been added to FGREP:

1. The basic unit that DICTION operates on is a sentence rather than a line. Each sentence that contains one of the patterns is output.
2. Upper case letters are mapped to lower case.
3. Punctuation is replaced by blanks.
4. All pattern matches in the sentence are found and surrounded with "[" "]"
5. A method for suppressing a string match has been added. Any pattern that begins with "~" will not be matched. Because the matching algorithm finds the longest substring, the suppression of a match allows words in some correct contexts not to be matched while allowing the word in another context to be found. For example, the word "which" is often incorrectly used instead of "that" in restrictive clauses. However, "which" is usually correct when preceded by a preposition or ",". The default pattern file suppresses the match of the common prepositions or a double blank followed by "which" and therefore matches only the suspect uses. The double blank accounts for the replaced comma.

8. Conclusions

A system of writing tools that measure some of the objective characteristics of writing style has been developed. The tools are sufficiently general that they may be applied to documents on any subject with equal accuracy. Although the measurements are only of the surface structure of the text, they do point out problem areas. In addition to helping writers produce better documents, these programs may be useful for studying the writing process and finding other formulae for measuring readability.

References

1. L. L. Cherry, "PARTS - A System for Assigning Word Classes to English Text," submitted *Communications of the ACM*.
2. B. W. Kernighan and J. R. Mashey, "The UNIX Programming Environment," *Software — Practice & Experience*, 9, 1-15 (1979).
3. G. R. Klare, "Assessing Readability," *Reading Research Quarterly*, 1974-1975, 10, 62-102.
4. E. A. Smith and P. Kincaid, "Derivation and validation of the automated readability index for use with technical materials," *Human Factors*, 1970, 12, 457-464.
5. J. P. Kincaid, R. P. Fishburne, R. L. Rogers, and B. S. Chissom, "Derivation of new readability formulas (Automated Readability Index, Fog count, and Flesch Reading Ease Formula) for Navy enlisted personnel," Navy Training Command Research Branch Report 8-75, Feb., 1975.
6. M. Coleman and T. L. Liau, "A Computer Readability Formula Designed for Machine Scoring," *Journal of Applied Psychology*, 1975, 60, 283-284.
7. R. Flesch, "A New Readability Yardstick," *Journal of Applied Psychology*, 1948, 32, 221-233.
8. E. U. Coke, private communication.
9. D. W. Ewing, *Writing for Results*, John Wiley & Sons, Inc., New York, N. Y. (1974).
10. G. Leggett, C. D. Mead and W. Charvat, *Prentice-Hall Handbook for Writers*, Seventh Edition, Prentice-Hall Inc., Englewood Cliffs, N. J. (1978).
11. E. B. Coleman, "Learning of Prose Written in Four Grammatical Transformations," *Journal of Applied Psychology*, 1965, vol. 49, no. 5, pp. 332-341.
12. A. V. Aho and M. J. Corasick, "Efficient String Matching: an aid to Bibliographic Search," *Communications of the ACM*, 18, (6), 333-340, June 1975.
13. Bell Laboratories, *"UNIX TIME-SHARING SYSTEM: UNIX PROGRAMMER'S MANUAL,"* Seventh Edition, Vol. 1 (January 1979).

Appendix 1

STYLE Abbreviations

a. d.
A. M.
a. m.
b. c.
Ch.
ch.
ckts.
dB.
Dept.
dept.
Depts.
depts.
Dr.
Drs.
e. g.
Eq.
eq.
et al.
etc.
Fig.
fig.
Figs.
figs.
ft.
i. e.
in.
Inc.
Jr.
jr.
mi.
Mr.
Mrs.
Ms.
No.
no.
Nos.
nos.
P. M.
p. m.
Ph. D.
Ph. d.
Ref.
ref.
Refs.
refs.
St.
vs.
yr.

Appendix 2

Default DICTION Patterns

a great deal of
a large number of
a lot of
a majority of
a need for
a number of
a particular preference for
a preference for
a small number of
a tendency to
above-mentioned
absolutely complete
absolutely essential
accompanied
accordingly
activate
actual
added increments
adequate enough
advent
afford an opportunity
aggregate
all of
all throughout
along the lines
an indication of
analyzation
and etc
and/or
another additional
any and all
arrive at a
as a matter of fact
as a method of
as good or better than
as of now
as per
as regards
as related to
as to
assistance
assistance to
assurance to
assuming that
at a later date
at about
at above
at all times
at an early date
at below
at the present
at the time when
at this point in time
at this time
at which time
at your earliest convenience
authorization
awful
basic fundamentals
basically
be cognizant of
being as
being that
brief in duration
bring to a conclusion
but that
but what
by means of
by the use of
carry out experiments
center about
center around
center portion
check into
check on
check up on
circle around
close proximity
collaborate together
collect together
combine together
come to an end
commence
common accord
communication
completely eliminated
comprise
concerning
conduct an investigation of
conjecture
connect up
consensus of opinion
consequent result
consolidate together
construct
contemplate
continue on
continue to remain
could of
count up
couple together
debate about
decide on
deleterious effect
demean
demonstrate
depreciate in value
deserving of
desirable benefits
desirous of
different than
discontinue
disutility
divide up
doubt but
due to
duly noted
during the time that
each and every
early beginnings
effectuate
emotional feelings
empty out
enclosed herein
enclosed herewith
end result
end up
endeavor
enter in
enter into
enthused
entirely complete
equally good as
essentially
eventuate
every now and then
exactly identical
experiencing difficulty
fabricate
face up to
facilitate
facts and figures
fast in action
fearful of
fearful that
few in number
file away
final completion
final ending
final outcome
final result
finalize
find it interesting to know
first and foremost
first beginnings
first initiated
firstly
follow after
following after
for the purpose of
for the reason that
for the simple reason that
for this reason
for your information
from the point of view of
full and complete
generally agreed
good and
got to
gratuitous
greatly minimize
head up
help but
hone in the production of
hopeful
if and when
if at all possible
impact
implement
important essentials
importantly
in a large measure
in a position to
in accordance
in advance of
in agreement with
in all cases
in back of
in behalf of
in behind
in between
in case
in close proximity
in conflict with
in conjunction with
in connection with
in fact
in large measure
in many cases
in most cases
in my opinion I think
in order to
in rare cases
in reference to
in regard to
in regards to
in relation with
in short supply
in size
in terms of
in the amount of
in the case of
in the course of
in the event
in the field of
in the form of
in the instance of
in the interim
in the last analysis
in the matter of
in the near future
in the neighborhood of
in the not too distant future
in the proximity of
in the range of
in the same way as described
in the sense of
in the vicinity of
in this case
in view of the
in violation of
inasmuch as
indicate
indicative of
initiate
innovate
injurious to
inquire
inside of
institute a
intents and purposes
intermingle
irregardless
is defined as
is used to control
is when
is where
it is incumbent
it stands to reason
it was noted that if
joint cooperation
joint partnership
just exactly
kind of
know about
last but not least
later on
leaving out of consideration
liable
link up
literally
little doubt that
lose out on
lots of
make allowance
make a
make adjustments to
make an
make application to
make contact with
make mention of
make out a list of
make the acquaintance of
make the adjustment
manner
maximum possible
meaningful
meet up with
melt down
melt up
methodology
might of
minimize as far as possible
minor importance
miss out on
modification

more preferable
most unique
must of
mutual cooperation
necessary requisite
necessitate
need for
[illegible]
not be as
not in a position to
not of a high order of accuracy
not un
notwithstanding
of considerable magnitude
of that
of the opinion that
off of
on a few occasions
on account of
on behalf of
on the grounds that
on the occasion
on the part of
one of the
open up
operates to correct
outside of
over with
overall
past history
perceptive of
perform a measurement
perform the measurement
permits the reduction of
personalize
pertaining to
physical size
plan ahead
plan for the future
plan in advance
plan on
present a conclusion
present a report
presently
prior to
prioritize
proceed to
procure
productive of
prolong the duration
protrude out from
provided that
pursuant to
put to use in
range all the way from
reason is because
reason why
recur again
reduce down
refer back
reference to this
reflective of
regarding
regretful
reiterate
relative to
repeat again
representative of
resultant effect
resume again
retreat back
return again
return back
revert back
seal off

seems apparent
send a communication
short space of time
should of
single unit
situation
so as to
sort of
spell out
still continue
still remain
subsequent
substantially in agreement
succeed in
suggestive of
superior than
surrounding circumstances
take appropriate
take cognizance of
take into consideration
termed as
terminate
termination
the author
the authors
the case that
the fact
the foregoing
the foreseeable future
the fullest possible extent
the majority of
the nature
the necessity of
the only difference being that
the order of
the point that
the truth is
there are not many
through the medium of
through the use of
throughout the entire
time interval
to summarize the above
total effect of all this
totality
transpire
true facts
try and
ultimate end
under a separate cover
under date of
under separate cover
under the necessity to
underlying purpose
undertake a study
uniformly consistent
unique
until such time as
up to this time
usage
utilize
very
very complete
very unique
vital
which
with a view to
with reference to
with regard to
with the exception of
with the object of
with the result that
with this in mind, it is clear that
within the realm of possibility
without further delay

worth while
would of
wrong behavior
wise
* which
* about which
* after which
* at which
* between which
* by which
* for which
* from which
* in which
* into which
* of which
* on which
* on which
* over which
* through which
* to which
* under which
* upon which
* with which
* without which
*clockwise
*likewise
*otherwise

Berkeley Font Catalog

October 1980

Introduction

This catalog gives samples of the various fonts available at Berkeley using vtroff on our Versatec and Varian. We have them working 4 pages across in a 36 inch Versatec, and rotated 90 degrees on a Benson-Varian 11 inch plotter. The same software should be adaptable to an 11 inch Versatec, and in fact is running at several other sites, however, not having one here, it isn't part of this distribution. Such a driver is available from Tom Ferrin at UCSF.

To use these fonts:

(1) Hershey. This is the default font. The Hershey font is currently the *only* complete font, with all 16 point sizes and all the special characters troff knows about. To get it, use vtroff directly. To illustrate this with the -ms macro package:

```
vtroff -ms paper.nr
```

(2) Fonts with roman, italic, and bold, such as nonie. You can load all three fonts with, for example:

```
vtroff -F nonie -ms paper.nr
```

To get just one of these fonts, use (3) below, appending .r, .i, or .b to the font name to specify which font you want mounted, e.g., to get italics in delegate,

```
vtroff -2 delegate.i -ms paper.nr
```

(3) To get a font without a complete set, choose which font (1, 2, or 3) you want replaced by the chosen font. For example, to use bocklin as though it were bold, since font 3 is bold, use:

```
vtroff -3 bocklin -ms paper.nr
```

To switch between fonts in troff, use

```
.ft 3
```

to switch to font 3, for example, or use

```
\f3word\f1
```

to switch within a line. For more information see the Nroff/Troff Users Manual.

Special note: *troff* thinks it is talking to a CAT phototypesetter. Thus, it does all sorts of strange things, such as enforcing restrictions like 7.54 inches maximum width, 4 fonts, a certain 18 point sizes, proportional spacing by point size, etc.

In particular, the following glyphs will *always* be taken from the special font, no matter what font you are using at the time:

@, #, ", ', `, <, >, \, {, }, ~, ^, and _

This may explain what are otherwise surprising results in some of the subsequent pages.

In addition, the following Greek letters have been decreed by *troff* as looking so much like their Roman counterparts that the Roman version (font 1) is always printed, no matter what font is mounted on font 1 at the time:

A, B, E, Z, H, I, K, M, N, O, P, T, X

(See table II in the back of the Nroff/Troff Users's Manual for details about what glyphs are in each font and how to generate the special glyphs.)

Font Layout Positions

Code	Normal	Special	Code	Normal	Special
000			100		
001			101	A	
002			102	B	
003			103	C	
004			104	D	
005			105	E	
006			106	F	
007			107	G	
010			110	H	
011			111	I	
012			112	J	
013			113	K	
014			114	L	
015			115	M	
016			116	N	
017			117	O	
020			120	P	
021			121	Q	
022			122	R	
023			123	S	
024			124	T	
025			125	U	
026			126	V	
027			127	W	
030			130	X	
031			131	Y	
032			132	Z	
033			133	[	
034			134	\	
035			135	]	
036			136	^	
037			137	_	
040			140	`	
041	!		141	a	
042	"		142	b	
043	#		143	c	
044	$		144	d	
045	%		145	e	
046	&		146	f	
047	'		147	g	
050	(		150	h	
051	)		151	i	
052	*		152	j	
053	+		153	k	
054	,		154	l	
055	-		155	m	
056	.		156	n	
057	/		157	o	
060	0		160	p	
061	1		161	q	
062	2		162	r	
063	3		163	s	
064	4		164	t	
065	5		165	u	
066	6		166	v	
067	7		167	w	
070	8		170	x	
071	9		171	y	
072	:		172	z	
073	;		173	{	
074	<		174	|	
075	=		175	}	
076	>		176	~	
077	?		177		

APL FONT, 10 POINT ONLY

A∝B⊥C∩D⊔E∈ F⌶G∇H∆I∪J⍳K' L⌷M⍱N⊤O○P⋆ Q?R⍴S⌽T~U↓ V∪W X⊃Y↑Z⊂ 01234 56789

(" # $ ∘ ⋆ ≥ ∨ ∧ ≤ ≠ • - { } { } ⌢ ~ _ \ ∃ ⍥ → < ∘ / \ . > , <

! → (⍁ → ⋆ & → ∘ ' → ' (→ ∨) → ∧ : → ≤ ⍟ → ≠ - → ∘ = → - [→ {] → } | → ⍪ → →
*; → < + → ∘ ? → *

Baskerville font, roman, bold, italic, 12 point only (Called "basker" on line)

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : ∘ - ⋆ [ ] { } ⌢ ~ _ \ | ⍥ ' ; ∘ / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : ∘ - ⋆ [ ] { } ⌢ ~ _ \ | ⍥ ' ; ∘ / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : ∘ - ⋆ [ ] { } ⌢ ~ _ \ | ⍥ ' ; ∘ / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Bocklin font. 14 and 28 point only.

14 point

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

" ( ) : - = [ ] ' ; / ? . .

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality: since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

28 point (No punctuation except period.)

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789 .

If time be of all things the most precious wasting time must be as Poor Richard says the greatest prodigality since as he elsewhere tells us lost time is never found again and what we call time enough always proves little enough Let us then up and be doing and doing to the purpose so by diligence shall we do more with less perplexity.

Bodoni font, roman, bold, *italic*, 10 point only.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : ; - = [ ] { } ^ ~ _ \ | @ ' ; * / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

*! " # $ % & ' ( ) : ; - = [ ] { } ^ ~ _ \ | @ ' ; * / ? . > , <*

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : ; - = [ ] { } ^ ~ _ \ | @ ' ; * / ? . > , <

1

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Chess, 18 point only

Note: Our attempt at compatibility with Stanford was only 99% successful. If you use a blank space to indicate an empty white square it will come out narrow due to the stupidity of troff. Either include the line

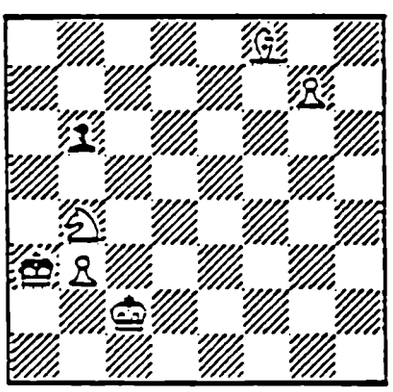
```
.cs ch 36
```

to put yourself in constant spacing mode or else use zero instead of space. You should also set the vertical spacing to 18 points.

```
.nf
.ft ch
.cs ch 36
.ps 18
.vs 18
HITTTTTTTX
V0Z0Z0A0ZF
VZ0Z0Z0C0F
V0o0Z0Z0ZF
VZ0Z0Z0Z0F
V0M0Z0Z0ZF
V]PZ0Z0Z0F
V0ZKZ0Z0ZF
VZ0Z0Z0Z0F
WUUUUUUUUG
.sp
.ft P
.ps 8
.cs P
```

p o b a n m r s q l k j U F G X 0

P O B A N M R S Q L K J T V W H Z



White mates in three moves.

Clarendon, 14 and 18 point roman only. From SAIL (Paul Martin & Andy Moorer)

ABCDE FGHIJ KLMNO PQRST UVWXY abcde fghij klmno pqrst uvwxyz 01234 56789

" # $ % × ' ( ) : - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXY abcde fghij klmno pqrst uvwxyz 01234 56789

" # $ % × ' ( ) : - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Computer Modern fonts, roman, *italic*, and bold. (by Don Knuth) 6, 7, 8, 9, 10, 11, 12 points. (Available as cm)

Note that the cm fonts are intended for TEX and don't fare so well with troff. The spacing is not proportional by point size, and hence only one point size can be tuned to be nicely spaced. We have tuned the 10 point size, but the 8 point looks somewhat cramped.

Some of the punctuation is missing in some of the fonts. Knuth also uses a nonstandard notion of ASCII, and hence some glyphs are available only with special symbols such as \(12. Others cannot be accessed at all.

Knuth's fonts somewhat larger than normal, since he intends the output to be reduced before printing. Since troff has a limitation of 7.5 inches width on output, this is not practical. Hence, the original fonts have been relabelled with the point size they are closest to without reduction. Some fonts (6 point bold, 7 point roman, 8 point italic and bold, 9 point bold, and 11 point italic) which would have otherwise been missing were generated by shrinking the next larger point size of the same style. (This goes against the idea of metafont, but we use the tools we have.)

10 Point Roman

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234
56789 ! " # ﬁ ﬂ ' ( ) * - { } ^ ~ _ \ ⊖ ÷ . > , < ' , Σ, ¬, T, Φ, Π, Δ, Θ, Λ, Ψ, Ω, ...

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality since, as he elsewhere tells us, lost time is never found again and what we call time enough, always proves little enough Let us then up and be doing, and doing to the purpose so by diligence shall we do more with less perplexity.

10 Point Italic

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234
*56789 ! " # ρ δ ϐ ' ( ) : * - = [ ] { } ^ ~ _ \ ω ⊖ ' ; + / ? . > , < ' , ' , Σ, ¬, ¬, Ξ, Υ, Φ, Π,*
η, ι, ı, Δ, Θ, Λ, Ψ, Ω, α, β, γ, ς ſ δ

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

10 Point Bold

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234
56789 ! " # ﬁ % & ' ( ) : * - = [ ] { } ^ ~ _ \ ﬂ ⊖ ' ; + / ? . > , < ' , ' , Σ, ¬, ¬, Ξ, Υ, Φ, Π,
¬, ¬, ¬, Δ, Θ, Λ, Ψ, Ω, ı, ȷ, ¬, ¬, ¬

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

6 Point Roman, **Bold**, and *Italic*
7 Point Roman, **Bold**, and *Italic*
8 Point Roman, **Bold**, and *Italic*
9 Point Roman, **Bold**, and *Italic*
10 Point Roman, **Bold**, and *Italic*
11 Point Roman, **Bold**, and *Italic*
12 Point Roman, **Bold**, and *Italic*

Countdown (22 point, upper case letters only.) From SAIL (Paul Martin)

ABCDE FGHIJ KLMNO PQRST UVWXYZ

COUNTDOWN HAS NO INTEGERS TO COUNT DOWN WITH BUT IT COMPENSATES BY BEING UGLY AND ILLEGIBLE

Cyrillic, 12 point only

ЖЦъЗ абде фгхи клмно прст увйз

ф тиме бе оф алл тхингс тхе мост прециоус вастинг тиме муст бе ас поор Ричард сайс тхе греатест продигалитй синце ас хе елсехере теллс ус лост тиме ис невер фоунд агаин анд хат е алл тиме еноугх алвайс провес литтле еноугх лет ус тхен уп анд бе доинг анд доинг то тхе пурпосе со бй дилигенце схалл е до море итх лесс перплекситй

W→Ж X→Ц Y→ъ Z→З a→а b→б d→д e→е f→ф g→г h→х i→и k→к l→л m→м n→н o→о p→п r→р s→с t→т u→у v→в y→й z→з

Delegate, roman, *italic*, and **bold**, 12 point only

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ? ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

*! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ / @ ; + / ? > , <*

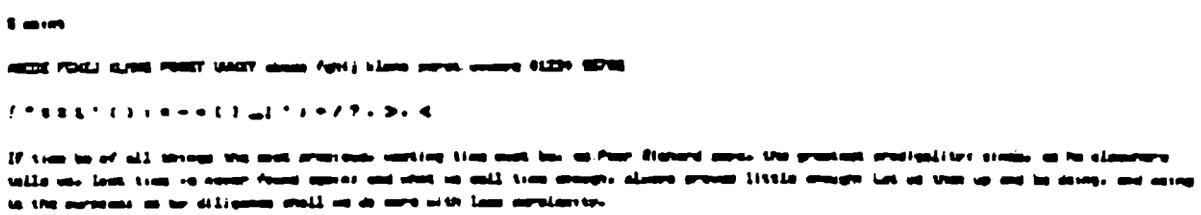
If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ↑ ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Fix fixed width font, 6, 9, 18, 12, 14 point



9 point

ABCDE FGHIJ KLMNO PQRST UVWXY abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

18 point

ABCDE FGHIJ KLMNO PQRST UVWXY abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

12 point

ABCDE FGHIJ KLMNO PQRST UVWXY abcde fghij klmno pqrst uvwxyz 81234 56789

! # $ % & ' ( ) * - = [ ] { } ~ ~ _ \ | @ ' : + / ? > . <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

14 point

ABCDE FGHIJ KLMNO PQRST UVWXY abcde fghij klmno pqrst uvwxyz 81234 56789

! " # $ % & ' ( ) : * - = [ ] { } ~ ~ _ \ | @ ' ; + / ? > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Gacham, roman, bold, *italic*, 18 point only
The gacham font is almost indistinguishable from the fix font. In fact, it has been pointed out that our gacham roman and bold fonts really are fix. Sign. They are included anyway for convenience.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) * + - = [ ] { } ^ ~ _ \ | @ ` ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality: since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789
*! " # $ % & ' ( ) * + - = [ ] { } ^ ~ _ \ | @ ` ; + / ? . > , <*

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789
! " # $ % & ' ( ) * + - = [ ] { } ^ ~ _ \ | @ ` ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Greek, 10 point only

This font provides an alternative to the Greek characters on the standard special font.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz

ΑΒΧΔΕ ΦΓΗΙϑ ΚΛΜΝΟ ΠΘΡΣΤ ΤΩΞΨΖ αβχδε φγηιϑ κλμνο πθρστ τωξψζ

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

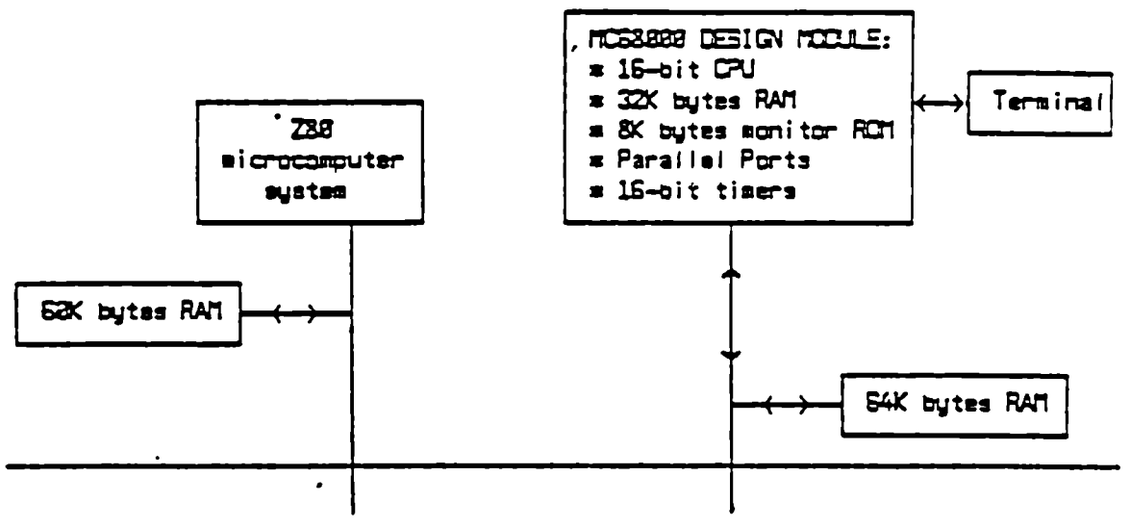
The h19 font includes a subset of the h19's graphic character set, plus a few logical extensions to allow forms and diagrams to be drawn. The characters are the same as the h19's graphic interpretation set.

a b c d e f g t u v w n h i k l

| ─ ┼ ┐ ┘ └ ┌ ┬ ┤ ┴ ├ ─ ┼ → ← ↓ ↑

The characters are designed to overlap.

Example of usage for diagrams:



Hebrew, 16, 24, and 38 point only

16 point

אבכדע עגהג בלמנ פקרשט תחחו עבש ך בלמ בש תחו 01234 56789

! " # ף צּ . ( ) : - [ ] { } ^ ~ _ \ | @ ּ ; ץ ? . > , <

עחם הדאהם בבלפ עבז נא עחפעללעבב עבנודם חלפ דנאלפורפ דערפעבבבלבב. פב האש עחם
אעאבנבאבע יפ עבנבב אלבלמב ונעדאבלע. לאמת ךן שב וע לל : ף עדם בבעב. : עב עאוחנ
ף עבב עלבב אעבל.

24 point

אבסד   גה ע בלמנ פקרשט ותחיו   ם

& → צ

תנ   רפ ר   ם הס מנ לל סח נא ש מנ ת דראהש המ
מש מלא נג ב   גאמנאודא המ שאה מ   .שג מס ד רפ
לבאדא רג

38 point (rather ragged)

אבסד   גה ע בלמנ פקרשט ותחיו

צ

10 point Hershey

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789 !, ?, &, ', (,), :, °, -, [,], ', :, /, ?, .

\(em → —, \– → –, \(bu → •, \(sq → ▪, \(ru → _ \(14 → ¼, \(12 → ½, \(34 → ¾, \(fi → fi, \(fl → fl, \(ff → ff, \(Fi → ffi, \(Fl → ffl, \(de → °, \(dg → †, \(fm → ′, \(ct → ¢\(rg → ®\(co → ©

When you flex your fingers in a coffin, it can baffle a giraffe.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789 !, ?, &, ', (,), :, °, -, [,], ', :, /, ?, .

\(em → —, \– → –, \(bu → •, \(sq → ▪, \(ru → _ \(14 → ¼\(12 → ½\(34 → ¾\(fi → fi, \(fl → fl, \(ff → ff, \(Fi → ffi, \(Fl → ffl, \(de → °, \(dg → †, \(fm → ′, \(ct → ¢\(rg → ®\(co → ©

When you flex your fingers in a coffin, it can baffle a giraffe.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789 !, ?, &, ', (,), :, °, -, [,], ', :, /, ?, .

\(em → —, \– → –, \(bu → •, \(sq → ▪, \(ru → _, \(14 → ¼\(12 → ½\(34 → ¾\(fi → fi, \(fl → fl, \(ff → ff, \(Fi → ffi, \(Fl → ffl, \(de → °, \(dg → †, \(fm → ′, \(ct → ¢\(rg → ®\(co → ©

When you flex your fingers in a coffin, it can baffle a giraffe.
From special font: " # = { } ^ ~ _ \ | @ ` ' + > <

Special characters: \(pl → +, \(mi → −, \(eq → =, \(** → *, \(sc → §, \(aa → ´, \(ga → `, \(ul → _, \(sl → /, \(*a → α, \(*b → β, \(*g → γ, \(*d → δ, \(*e → ε, \(*z → ζ, \(*y → η, \(*h → θ, \(*i → ι, \(*k → κ, \(*l → λ, \(*m → μ, \(*n → ν, \(*c → ξ, \(*o → ο, \(*p → π, \(*r → ρ, \(*s → σ, \(ts → ς, \(*t → τ, \(*u → υ, \(*f → φ, \(*x → χ, \(*q → ψ, \(*w → ω, \(*A → Α, \(*B → Β, \(*G → Γ, \(*D → Δ, \(*E → Ε, \(*Z → Ζ, \(*Y → Η, \(*H → Θ, \(*I → Ι, \(*K → Κ, \(*L → Λ, \(*M → Μ, \(*N → Ν, \(*C → Ξ, \(*O → Ο, \(*P → Π, \(*R → Ρ, \(*S → Σ, \(*T → Τ, \(*U → Υ, \(*F → Φ, \(*X → Χ, \(*Q → Ψ, \(*W → Ω, \(sr → √, \(rn → ‾, \(>= → ≥, \(<= → ≤, \(== → ≡, \(~= → ≅, \(ap → ~, \(!= → ≠, \(-> → →, \(<- → ←, \(ua → ↑, \(da → ↓, \(mu → ×, \(di → ÷, \(+- → ±, \(cu → ∪, \(ca → ∩, \(sb → ⊂, \(sp → ⊃, \(ib → ⊆, \(ip → ⊇, \(if → ∞, \(pd → ∂, \(gr → ∇, \(no → ¬, \(is → ∫, \(pt → ∝, \(eq → ∈, \(no → ∉, \(br → ¦, \(dd → ‡, \(rh → ⇀, \(lh → ↽, \(bs → \, \(or → |, \(ci → ○, \(lt → {, \(lb → ⌊, \(rt → ⌋, \(rb → ⌋, \(lk → {, \(rk → }, \(bv → |, \(lf → ⌊, \(rf → ⌋, \(lc → ⌈, \(rc → ⌉

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality: since, as he elsewhere tells us, lost time is never found again: and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

This is an *example* of a sample in various fonts.

Hershey font. This is the default font for ntroff. Roman, *Italic* and Bold in 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, and 36 point. The following examples are 10 point.

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

6 point Roman, Bold, and *Italic*.
7 point Roman, Bold, and *Italic*.
8 point Roman, Bold, and *Italic*.
9 point Roman, Bold, and *Italic*.
10 point Roman, Bold, and *Italic*.
11 point Roman, Bold, and *Italic*.
12 point Roman, Bold, and *Italic*.
14 point Roman, Bold, and *Italic*.
16 point Roman, Bold, and *Italic*.
18 point Roman, Bold, and *Italic*.
20 point Roman, Bold, and *Italic*.
22 point Roman, Bold, and *Italic*.
24 point Roman, Bold, and *Italic*.
28 point Roman, Bold, and *Italic*.
36 point Roman, Bold, and *Italic*

Meteor, roman, bold, *italic*, 8, 10, 12 point, no 12 point italic.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ~ ^ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

*! " # $ % & ' ( ) : * - = [ ] { } ~ ^ _ \ | @ ' ; + / ? . > , <*

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ~ ^ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Microgramma font, 10 point only

ABCDE FGHIJ KLMNO PQRST UVWXY abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : ; - + [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most pracious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Mona font, 24 point only

ABCDE FGHIJ KLMNO PQRST UVWXYZ
abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ ¢ & ' ( ) : - { } ^ ~ _ \ @ ; ? . > , <

Philadelphia is the most pecksniffian of American cities, and thos probably leads the world.

- H. L. Mencken

None, roman, bold, *italic*, 8, 10, 12 point

8 point
ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

*! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <*

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

10 point
ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

*! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <*

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

12 point
ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : ² - ³ [ ] { } ~ ~ _ \ | © ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : ² - ³ [ ] { } ~ ~ _ \ | © ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : ² - ³ [ ] { } ~ ~ _ \ | © ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Old English, 8, 14, and 18 point only. (This font is called "oldenglish" on line.)

8 point

ABCDE FGHIJ KLMNO PQRST UVWXY abcde fghij klmno pqrst uvwxyz 01234 56789

" # : { } ^ ~ _ \ @ : .> .<

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality since, as he elsewhere tells us, lost time is never found again and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose: so by diligence shall we do more with less perplexity.

14 point

ABCDE FGHIJ KLMNO PQRST UVWXY abcde fghij klmno pqrst uvwxyz 01234 56789

" # : { } ^ ~ _ \ @ : .> .<

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality;since, as he elsewhere tells us, lost time is never found again;and what we call time enough, always proves little enough:Let us then up and be doing, and doing to the purpose;so by diligence shall we do more with less perplexity.

18 point

ABCDE FGHIJ KLMNO PQRST UVWXYZ
abcde fghij klmno pqrst uvwxyz 01234 56789

" # ' ( ) - { } ^ ~ - \ ? . > . <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality since, as he elsewhere tells us, lost time is nver found again and what we call time enough, always proves little enough and I think I'm wasting time typing all this stuff

PIP FONT, 16 POINT ONLY, NO LOWER CASE

ABCDE FGHIJ KLMNO PQRST UVWXYZ 01234 56789

!" # '( ): - { } ^ ~ _\ @ '; ?. >, <

IT COULD PROBABLY BE SHOWN BY FACTS AND FIGURES THAT THERE IS NO DISTINCTLY NATIVE AMERICAN CRIMINAL CLASS EXCEPT CONGRESS.

– MARK TWAIN

Plain font, 18 point only

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

!" # $ % & ' ( ) : * - + [ ] { } ^ ~ _\ @ ; + / ? . >, <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Script, 18 point only. This font appears to be almost identical to the "Coronet" font from SAIL, except that the period and one other glyph of Coronet are missing a row, and Coronet is supposed to be 16 point. (They are both really the same size.)

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

" # { } ^ ~ _\ @ >, <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

SHADOW, 16 POINT ONLY, NO LOWER CASE

ABCDE FGHIJ KLMNO PQRST UVWXYZ 01234 56789

! " # ' , [ ] { } ^ ~ _ \ @ ‘ ; ◇ . > , <

THE SHADOW FONT IS AN EXCELLENT CHOICE FOR PROFOUND PREDICTIONS. IT HAS THE ADVANTAGE OF BEING ALMOST UNREADABLE.

◇◇ S H A D O W ◇◇

SIGN, 22 POINT ONLY

ABCDE FGHIJ KLMNO PQRST
UVWXYZ ➢➣ 0123456789

! " # ' :* - = { } ^ ~ _ @ ; / . > , <

THIS FONT WAS INVENTED BY A DRAFTSMAN WHO HAD LOST HIS FRENCH CURVE. ➢SO IT GOES ➣

LOWER CASE L IS ➢, LOWER CASE R IS ➣.

Stare hershey font. This font is identical to the hershey font except that the point sizes are one point smaller, and the width tables are those used for the real typesetter. Hence, this font is useful when previewing documents that are to be sent to a typesetter to make sure the spacing, paging, and so on is right. There are Roman, *Italic* and **Bold** in 8, 9, 10, 11, 12, 14, and 16 point. The following examples are 10 point.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ~ ^ _ \ | @ ` ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

*! " # $ % & ' ( ) : * - = [ ] { } ~ ^ _ \ | @ ` ; + / ? . > , <*

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ~ ^ _ \ | @ ` ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

8 point Roman, **Bold**, and *Italic*.
9 point Roman, **Bold**, and *Italic*.
10 point Roman, **Bold**, and *Italic*.
11 point Roman, **Bold**, and *Italic*.
12 point Roman, **Bold**, and *Italic*.
14 point Roman, **Bold**, and *Italic*.
16 point Roman, **Bold**, and *Italic*.

Times fonts, roman, *italic*, and bold. 10 point only.
These fonts showed up in a directory labelled "timesroman" along with three other fonts which turned out to be nonie, meteor, and news gothic. They are probably not really times fonts, but seem to be pretty close. Notice the top of the "2" for a clear difference from a real Times Roman font.

It is our desire to have a real, digitized version of the times fonts from the phototypesetter. We eventually plan to do this. At that point, the times font will probably replace the hershey font as the default. Such a Times font is already available from Johns Hopkins University for a fee, but we couldn't redistribute it, so we plan do digitize them ourselves.

10 Point

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789
! " # $ % & ' ( ) : ; - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <
', ', —, -, -, -, •, □, , ¼, ½, ¾, fi, fl, ff, ffi, ffl, °, †, ', ¶, ©

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789
! " # $ % & ' ( ) : ; - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <
', ', —, -, -, -, •, □, , ¼, ½, ¾, fi, fl, ff, ffi, ffl, °, †, ', ¶, ©

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789
! " # $ % & ' ( ) : ; - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <
', ', —, -, -, -, •, □, , ¼, ½, ¾, fi, fl, ff, ffi, ffl, °, †, ', ¶, ©

NAME

font names – table of font names in short and long formats

SYNOPSIS

cat /usr/lib/fontinfo/kurz

DESCRIPTION

For the usage of fonts other than the default ones in *troff* (or *ltroff* or *vtroff* resp.) the names of these fonts must be specified twice. The full name (see below) is used to control the phototypesetter or the postprocessor (*lcat* or *vcat*). *Troff* itself needs the specification of the font name in a short form for the selection of the corresponding font size tables in a **.fp** -command.

long name	short name	long name	short name
apl	ap	h19	hn
basker.b	bb	hebrew	hb
basker.i	bi	meteor.b	mb
basker.r	br	meteor.i	mi
bocklin	bk	meteor.r	mr
bodoni.b	ob	mona	mn
bodoni.i	oi	nonie.b	nb
bodoni.r	or	nonie.i	ni
chess	ch	nonie.r	nr
clarendon	cl	oldenglish	oe
cm.b	cb	pip	pp
cm.i	ci	playbill	pb
cm.r	cr	script	sc
countdown	co	shadow	sh
cyrillic	cy	sign	sg
delegate.b	db	stare.b	sb
delegate.i	di	stare.i	si
delegate.r	dr	stare.r	sr
fix	fx	times.b	tb
gacham.b	gb	times.i	ti
gacham.i	gi	times.r	tr
gacham.r	gr	times.s	ts
graphics	gf	ugramma	m
greek	gk		

FILES

/usr/lib/fontinfo/kurz

NAME

font list – table of available fonts and point sizes

DESCRIPTION

font	available sizes									
R	6	7	8	9	10	11	12	14	16	18
	20	22	24	28	36					
B	6	7	8	9	10	11	12	14	16	18
	20	22	24	28	36					
I	6	7	8	9	10	11	12	14	16	18
	20	22	24	28	36					
S	6	7	8	9	10	11	12	14	16	18
	20	22	24	28	36					
apl	10									
basker.r	12									
basker.b	12									
basker.i	12									
bocklin	14	28								
bodoni.r	10									
bodoni.b	10									
bodoni.i	10									
chess	18									
clarendon	14	18								
cm.r	6	7	8	9	10	11	12			
cm.b	6	7	8	9	10	11	12			
cm.i	6	7	8	9	10	11	12			
countdown	22									
cyrillic	12									
delegate.r	12									
delegate.b	12									
delegate.i	12									
fix	6	9	10	12	14					
gacham.r	10									
gacham.b	10									
gacham.i	10									
graphics	14									
greek	10									
h19	10									
hebrew	16	17	24	36						
meteor.r	8	10	12							
meteor.b	8	10	12							
meteor.i	8	10								
mona	24									
nonie.r	8	10	12							
nonie.b	8	10	12							
nonie.i	8	10	12							

oldenglish	8	14	18				
pip	16						
playbill	10						
script	18						
shadow	16						
sign	22						
stare.r	8	9	10	11	12	14	16
stare.b	8	9	10	11	12	14	16
stare.i	8	9	10	11	12	14	16
times.r	10						
times.b	10						
times.i	10						
times.s	10						
ugramma	10						

Screen Updating and Cursor Movement Optimization:

A Library Package

This document describes a package of C library functions which allow the user to:

update a screen with reasonable optimization,

get input from the terminal in a screen-oriented fashion, and

independent from the above, move the cursor optimally from one point to another.

These routines all use the **/etc/termcap** database to describe the capabilities of the terminal.

Acknowledgements

This package would not exist without the work of Bill Joy, who, in writing his editor, created the capability to generally describe terminals, wrote the routines which read this database, and, most importantly, those which implement optimal cursor movement, which routines I have simply lifted nearly intact. Doug Merrit and Kurt Shoens also were extremely important, as were both willing to waste time listing to me rant and rave. The help and/or support of Ken Abrams, Alan Char, Mark Horton, and Joe Kalash, was, and is, also greatly appreciated.

Kenneth C. R. C. Arnold

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

Screen Package

Contents

Appendices

1. Overview

In making available the generalized terminal descriptions in /etc/termcap, much information was made available to the programmer, but little work was taken out of one's hands. The purpose of this package is to allow the C programmer to do the most common type of terminal dependent functions, those of movement optimization and optimal screen updating, without doing any of the dirty work, and (hopefully) with nearly as much ease as is necessary to simply print or read things.

The package is split into three parts: (1) Screen updating; (2) Screen updating with user input; and (3) Cursor motion optimization.

It is possible to use the motion optimization without using either of the other two, and screen updating and input can be done without any programmer knowledge of the motion optimization, or indeed the database itself.

1.1. Terminology (or, Words You Can Say to Sound Brilliant)

In this document, the following terminology is kept to with reasonable consistency:

window: An internal representation containing an image of what a section of the terminal screen may look like at some point in time. This subsection can either encompass the entire terminal screen, or any smaller portion down to a single character within that screen.

terminal: Sometimes called *terminal screen*. The package's idea of what the terminal's screen currently looks like, i.e., what the user sees now. This is a special *screen*.

screen: This is a subset of windows which are as large as the terminal screen, i.e., they start at the upper left hand corner and encompass the lower right hand corner. One of these, *stdscr*, is automatically provided for the programmer.

1.2. Compiling Things

In order to use the library, it is necessary to have certain types and variables defined. Therefore, the programmer must have a line:

```
#include <curses.h>
```

at the top of the program source. The header file <curses.h> needs to include <sgtty.h>, so the one should not do so oneself[1]. Also, compilations should have the following form:

```
cc [ flags ] file ... -lcurses -ltermlib
```

1.3. Screen Updating

In order to update the screen optimally, it is necessary for the routines to know what the screen currently looks like and what the programmer wants it to look like next. For this purpose, a data type (structure) named *WINDOW* is defined which describes a window image to the routines, including its starting position on the screen (the (y, x) co-ordinates of the upper left hand corner) and its size. One of these (called *curscr* for *current screen*) is a screen image of what the terminal currently looks like. Another screen (called *stdscr*, for *standard screen*) is provided by default to make changes on.

A window is a purely internal representation. It is used to build and store a potential image of a portion of the terminal. It doesn't bear any necessary relation to what is really on the terminal screen. It is more like an array of characters on which to make changes.

When one has a window which describes what some part the terminal should look like, the routine *refresh()* (or *wrefresh()* if the window is not *stdscr*) is called. *refresh()* makes the ter-

[1] The screen package also uses the Standard I/O library, so <curses.h> includes <stdio.h>. It is redundant (but harmless) for the programmer to do it, too.

minal, in the area covered by the window, look like that window. Note, therefore, that changing something on a window *does not change the terminal.* Actual updates to the terminal screen are made only by calling *refresh()* or *wrefresh()*. This allows the programmer to maintain several different ideas of what a portion of the terminal screen should look like. Also, changes can be made to windows in any order, without regard to motion efficiency. Then, at will, the programmer can effectively say "make it look like this," and let the package worry about the best way to do this.

1.4. Naming Conventions

As hinted above, the routines can use several windows, but two are automatically given: *curscr*, which knows what the terminal looks like, and *stdscr*, which is what the programmer wants the terminal to look like next. The user should never really access *curscr* directly. Changes should be made to the appropriate screen, and then the routine *refresh()* (or *wrefresh()*) should be called.

Many functions are set up to deal with *stdscr* as a default screen. For example, to add a character to *stdscr*, one calls *addch()* with the desired character. If a different window is to be used, the routine *waddch()* (for window-specific *addch()*) is provided[2]. This convention of prepending function names with a "w" when they are to be applied to specific windows is consistent. The only routines which do *not* do this are those to which a window must always be specified.

In order to move the current (y, x) co-ordinates from one point to another, the routines *move()* and *wmove()* are provided. However, it is often desirable to first move and then perform some I/O operation. In order to avoid clumsyness, most I/O routines can be preceded by the prefix "mv" and the desired (y, x) co-ordinates then can be added to the arguments to the function. For example, the calls

```
move(y, x);
addch(ch);
```

can be replaced by

```
mvaddch(y, x, ch);
```

and

```
wmove(win, y, x);
waddch(win, ch);
```

can be replaced by

```
mvwaddch(win, y, x, ch);
```

Note that the window description pointer (*win*) comes before the added (y, x) co-ordinates. If such pointers are need, they are always the first parameters passed.

2. Variables

Many variables which are used to describe the terminal environment are available to the programmer. They are:

type	name	description
WINDOW *	curscr	current version of the screen (terminal screen).
WINDOW *	stdscr	standard screen. Most updates are usually done here.
char *	Def_term	default terminal type if type cannot be determined

[2] Actually, *addch()* is really a "#define" macro with arguments, as are most of the "functions" which deal with *stdscr* as a default.

bool	My_term	use the terminal specification in *Def_term* as terminal, irrelevant of real terminal type
char *	ttytype	full name of the current terminal.
int	LINES	number of lines on the terminal
int	COLS	number of columns on the terminal
int	ERR	error flag returned by routines on a fail.
int	OK	error flag returned by routines when things go right.

There are also several "#define" constants and types which are of general usefulness:

reg	storage class "register" (e.g., *reg int i;*)
bool	boolean type, actually a "char" (e.g., *bool doneit;*)
TRUE	boolean "true" flag (1).
FALSE	boolean "false" flag (0).

3. Usage

This is a description of how to actually use the screen package. In it, we assume all updating, reading, etc. is applied to *stdscr*. All instructions will work on any window, with changing the function name and parameters as mentioned above.

3.1. Starting up

In order to use the screen package, the routines must know about terminal characteristics, and the space for *curscr* and *stdscr* must be allocated. These functions are performed by *initscr()*. Since it must allocate space for the windows, it can overflow core when attempting to do so. On this rather rare occasion, *initscr()* returns ERR. *initscr()* must *always* be called before any of the routines which affect windows are used. If it is not, the program will core dump as soon as either *curscr* or *stdscr* are referenced. However, it is usually best to wait to call it until after you are sure you will need it, like after checking for startup errors. .Terminal status changing routines like *nl()* and *crmode()* should be called after *initscr()*.

Now that the screen windows have been allocated, you can set them up for the run. If you want to, say, allow the window to scroll, use *scrollok()*. If you want the cursor to be left after the last change, use *leaveok()*. If this isn't done, *refresh()* will move the cursor to the window's current (y, x) co-ordinates after updating it. New windows of your own can be created, too, by using the functions *newwin()* and *subwin()*. *delwin()* will allow you to get rid of old windows. If you wish to change the official size of the terminal by hand, just set the variables *LINES* and *COLS* to be what you want, and then call *initscr()*. This is best done before, but can be done either before or after, the first call to *initscr()*, as it will always delete any existing *stdscr* and/or *curscr* before creating new ones.

3.2. The Nitty-Gritty

3.2.1. Output

Now that we have set things up, we will want to actually update the terminal. The basic functions used to change what will go on a window are *addch()* and *move()*. *addch()* adds a character at the current (y, x) co-ordinates, returning ERR if it would cause the window to illegally scroll, i.e., printing a character in the lower right-hand corner of a terminal which automatically scrolls if scrolling is not allowed. *move()* changes the current (y, x) co-ordinates to whatever you want them to be. It returns ERR if you try to move off the window when scrolling is not allowed. As mentioned above, you can combine the two into *mvaddch()* to do both things in one fell swoop.

The other output functions, such as *addstr()* and *printw()*, all call *addch()* to add characters to the window.

After you have put on the window what you want there, when you want the portion of the terminal covered by the window to be made to look like it, you must call *refresh()*. In order

to optimize finding changes, *refresh()* assumes that any part of the window not changed since the last *refresh()* of that window has not been changed on the terminal, i.e., that you have not refreshed a portion of the terminal with an overlapping window. If this is not the case, the routine *touchwin()* is provided to make it look like the entire window has been changed, thus making *refresh()* check the whole subsection of the terminal for changes.

If you call *wrefresh()* with *curscr*, it will make the screen look like *curscr* thinks it looks like. This is useful for implementing a command which would redraw the screen in case it gets messed up.

3.2.2. Input

Input is essentially a mirror image of output. The complementary function to *addch()* is *getch()* which, if echo is set, will call *addch()* to echo the character. Since the screen package needs to know what is on the terminal at all times, if characters are to be echoed, the tty must be in raw or cbreak mode. If it is not, *getch()* sets it to be cbreak, and then reads in the character.

3.2.3. Miscellaneous

All sorts of fun functions exists for maintaining and changing information about the windows. For the most part, the descriptions in section 5.4. should suffice.

3.3. Finishing up

In order to do certain optimizations, and, on some terminals, to work at all, some things must be done before the screen routines start up. These functions are performed in *gettmode()* and *setterm()*, which are called by *initscr()*. In order to clean up after the routines, the routine *endwin()* is provided. It restores tty modes to what they were when *initscr()* was first called. Thus, anytime after the call to initscr, *endwin()* should be called before exiting.

4. Cursor Motion Optimization: Standing Alone

It is possible to use the cursor optimization functions of this screen package without the overhead and additional size of the screen updating functions. The screen updating functions are designed for uses where parts of the screen are changed, but the overall image remains the same. This includes such programs as eye and vi[3]. Certain other programs will find it difficult to use these functions in this manner without considerable unnecessary program overhead. For such applications, such as some "*crt hacks*"[4] and optimizing cat(1)-type programs, all that is needed is the motion optimizations. This, therefore, is a description of what some of what goes on at the lower levels of this screen package. The descriptions assume a certain amount of familiarity with programming problems and some finer points of C. None of it is terribly difficult, but you should be forewarned.

4.1. Terminal Information

In order to use a terminal's features to the best of a program's abilities, it must first know what they are[5]. The /etc/termcap database describes these, but a certain amount of decoding is necessary, and there are, of course, both efficient and inefficient ways of reading them in. The algorithm that the uses is taken from vi and is hideously efficient. It reads them in a tight loop into a set of variables whose names are two uppercase letters with some mnemonic value. For

---

[3] Eye actually uses these functions, vi does not.

[4] Graphics programs designed to run on character-oriented terminals. I could name many, but they come and go, so the list would be quickly out of date. Recently, there have been programs such as rocket and gun.

[5] If this comes as any surprise to you, there's this tower in Paris they're thinking of junking that I can let you have for a song.

example, *HO* is a string which moves the cursor to the "home" position[6]. As there are two types of variables involving ttys, there are two routines. The first, *gettmode()*, sets some variables based upon the tty modes accessed by gtty(2) and stty(2) The second, *setterm()*, a larger task by reading in the descriptions from the /etc/termcap database. This is the way these routines are used by *initscr()*:

```
If (isatty(0)) {
     gettmode();
     if (sp=getenv("TERM"))
          setterm(sp);
}
else
     setterm(Def_term);
_puts(TI);
_puts(VS);
```

isatty() checks to see if file descriptor 0 is a terminal[7]. If it is, *gettmode()* sets the terminal description modes from a gtty(2) *getenv()* is then called to get the name of the terminal, and that value (if there is one) is passed to *setterm()*, which reads in the variables from /etc/termcap associated with that terminal. (*getenv()* returns a pointer to a string containing the name of the terminal, which we save in the character pointer *sp*.) If *isatty()* returns false, the default terminal *Def_term* is used. The *TI* and *VS* sequences initialize the terminal (*_puts()* is a macro which uses *tputs()* (see termcap(3)) to put out a string). It is these things which *endwin()* undoes.

4.2. Movement Optimizations, or, Getting Over Yonder

Now that we have all this useful information, it would be nice to do something with it[8] The most difficult thing to do properly is motion optimization. When you consider how many different features various terminals have (tabs, backtabs, non-destructive space, home sequences, absolute tabs, ...) you can see that deciding how to get from here to there can be a decidedly non-trivial task. The editor vi uses many of these features, and the routines it uses to do this take up many pages of code. Fortunately, I was able to liberate them with the author's permission, and use them here.

After using *gettmode()* and *setterm()* to get the terminal descriptions, the function *mvcur()* deals with this task. It usage is simple: you simply tell it where you are now and where you want to go. For example

```
mvcur(0, 0, LINES/2, COLS/2)
```

would move the cursor from the home position (0, 0) to the middle of the screen. If you wish to force absolute addressing, you can use the function *tgoto()* from the termlib(7) routines, or you can tell *mvcur()* that you are impossibly far away, like Cleveland. For example, to absolutely address the lower left hand corner of the screen from anywhere just claim that you are in the upper right hand corner:

```
mvcur(0, COLS-1, LINES-1, 0)
```

---

[6] These names are identical to those variables used in the /etc/termcap database to describe each capability. See Appendix A for a complete list of those read, and termcap(5) for a full description.

[7] *isatty()* is defined in the default C library function routines. It does a gtty(2) on the descriptor and checks the return value.

[8] Actually, it *can* be emotionally fulfilling just to get the information. This is usually only true, however, if you have the social life of a kumquat.

5. The Functions

In the following definitions, "†" means that the "function" is really a "#define" macro with arguments. This means that it will not show up in stack traces in the debugger, or, in the case of such functions as *addch()*, it will show up as it's "w" counterpart. The arguments are given to show the order and type of each. Their names are not mandatory, just suggestive.

5.1. Output Functions

```
addch(ch) †
char    ch;
```

```
waddch(win, ch)
WINDOW  *win;
char    ch;
```

Add the character *ch* on the window at the current (y, x) co-ordinates. If the character is a newline ('\n') the line will be cleared to the end, and the current (y, x) co-ordinates will be changed to the beginning off the next line if newline mapping is on, or to the next line at the same x co-ordinate if it is off. A return ('\r') will move to the beginning of the line on the window. Tabs ('\t') will be expanded into spaces in the normal tabstop positions of every eight characters. This returns ERR if it would cause the screen to scroll illegally.

```
addstr(str) †
char    *str;
```

```
waddstr(win, str)
WINDOW  *win;
char    *str;
```

Add the string pointed to by *str* on the window at the current (y, x) co-ordinates. This returns ERR if it would cause the screen to scroll illegally. In this case, it will put on as much as it can.

```
box(win, vert, hor)
WINDOW  *win;
char    vert, hor;
```

Draws a box around the window using *vert* as the character for drawing the vertical sides, and *hor* for drawing the horizontal lines. If scrolling is not allowed, and the window encompasses the lower right-hand corner of the terminal, the corners are left blank to avoid a scroll.

```
clear() †
```

```
wclear(win)
WINDOW  *win;
```

Resets the entire window to blanks. If *win* is a screen, this sets the clear flag, which will cause a clear-screen sequence to be sent on the next *refresh()* call. This also moves the current (y, x) co-ordinates to (0, 0).

clearok(scr, boolf) †
WINDOW *scr;
bool boolf;

Sets the clear flag for the screen *scr*. If *boolf* is TRUE, this will force a clear-screen to be printed on the next *refresh()*, or stop it from doing so if *boolf* is FALSE. This only works on screens, and, unlike *clear()*, does not alter the contents of the screen. If *scr* is *curscr*, the next *refresh()* call will cause a clear-screen, even if the window passed to *refresh()* is not a screen.

clrtobot() †

wclrtobot(win)
WINDOW *win;

Wipes the window clear from the current (y, x) co-ordinates to the bottom. This does not force a clear-screen sequence on the next refresh under any circumstances. This has no associated "mv" command.

clrtoeol() †

wclrtoeol(win)
WINDOW *win;

Wipes the window clear from the current (y, x) co-ordinates to the end of the line. This has no associated "mv" command.

delch()

wdelch(win)
WINDOW *win;

Delete the character at the current (y, x) co-ordinates. Each character after it on the line shifts to the left, and the last character becomes blank.

deleteln()

wdeleteln(win)
WINDOW *win;

Delete the current line. Every line below the current one will move up, and the bottom line will become blank. The current (y, x) co-ordinates will remain unchanged.

erase() †

werase(win)
WINDOW *win;

Erases the window to blanks without setting the clear flag. This is analagous to *clear()*, except that it never causes a clear-screen sequence to be generated on a *refresh()*. This has no associated "mv" command.

insch(c)
char c;

winsch(win, c)
*WINDOW *win;*
char c;

Insert *c* at the current (y, x) co-ordinates Each character after it shifts to the right, and the last character disappears.

insertln()

winsertln(win)
*WINDOW *win;*

Insert a line above the current one. Every line below the current line will be shifted down, and the bottom line will disappear. The current line will become blank, and the current (y, x) co-ordinates will remain unchanged.

move(y, x) †
int y, x;

wmove(win, y, x)
*WINDOW *win;*
int y, x;

Change the current (y, x) co-ordinates of the window to (*y, x*). This returns ERR if it would cause the screen to scroll illegally.

overlay(win1, win2)
*WINDOW *win1, *win2;*

Overlay *win1* on *win2*. The contents of *win1*, insofar as they fit, are placed on *win2* at their starting (y, x) co-ordinates. This is done non-destructively, i.e., blanks on *win1* leave the contents of the space on *win2* untouched.

overwrite(win1, win2)
*WINDOW *win1, *win2;*

Overwrite *win1* on *win2*. The contents of *win1*, insofar as they fit, are placed on *win2* at their starting (y, x) co-ordinates. This is done destructively, i.e., blanks on *win1* become blank on *win2*.

printw(fmt, arg1, arg2, ...)
*char *fmt;*

wprintw(win, fmt, arg1, arg2, ...)
WINDOW **win;*
char **fmt;*

Performs a *printf()* on the window starting at the current (y, x) co-ordinates. It uses *addstr()* to add the string on the window. It is often advisable to use the field width options of *printf()* to avoid leaving things on the window from earlier calls. This returns ERR if it would cause the screen to scroll illegally.

refresh() †

wrefresh(win)
WINDOW **win;*

Synchronize the terminal screen with the desired window. If the window is not a screen, only that part covered by it is updated. This returns ERR if it would cause the screen to scroll illegally. In this case, it will update whatever it can without causing the scroll.

standout() †

wstandout(win)
WINDOW **win;*

standend() †

wstandend(win)
WINDOW **win;*

Start and stop putting characters onto *win* in standout mode. *standout()* causes any characters added to the window to be put in standout mode on the terminal (if it has that capability). *standend()* stops this. The sequences *SO* and *SE* (or *US* and *UE* if they are not defined) are used (see Appendix A).

5.2. Input Functions

crmode() †

nocrmode() †

Set or unset the terminal to/from cbreak mode.

echo() †

noecho() †

Sets the terminal to echo or not echo characters.