UNIX PROGRAMMER'S MANUAL

4.2 Berkeley Software Distribution, Volume 2c
Virtual VAX−11 Version

August, 1983

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

Copyright 1979, 1980 Regents of the University of California. Permission to copy these documents or any portion thereof as necessary for licensed use of the software is granted to licensees of this software, provided this copyright notice and statement of permission are included.

The document "Writing Tools – The STYLE and DICTION Programs" is copyrighted 1979 by Bell Telephone Laboratories. Holders of a UNIX™/32V software license are permitted to copy this document, or any portion of it, as necessary for licensed use of the software, provided this copyright notice and statement of permission are included.

The document "The Programming Language EFL" is copyrighted 1979 by Bell Telephone Laboratories. EFL has been approved for general release, so that one may copy it subject only to the restriction of giving proper acknowledgement to Bell Telephone Laboratories.

The documents "A Portable Fortran 77 Compiler" and "Fsck – The UNIX File System Check Program" are modifications of earlier documents which are copyrighted 1979 by Bell Telephone Laboratories. Holders of a UNIX™/32V software license are permitted to copy these documents, or any portion of them, as necessary for licensed use of the software, provided this copyright notice and statement of permission are included.

This manual reflects system enhancements made at Berkeley and sponsored in part by NSF Grants MCS-7807291, MCS-8005144, and MCS-74-07644-A04; DOE Contract DE-AT03-76SF00034 and Project Agreement DE-AS03-79ER10358; and by Defense Advanced Research Projects Agency (DoD) ARPA Order No. 4031, Monitored by Naval Electronics Systems Command under Contract No. N00039-80-K-0649.

UNIX Programmer's Manual

Volume 2c — Supplementary Documents

4.2 Berkeley Software Distribution, Virtual VAX−11 Version

August, 1983

This volume contains documents which supplement the information in Volume 1 of *The UNIX† Programmer's Manual*, for the Virtual VAX-11 version of the system as distributed by U.C. Berkeley. This volume is a logical extension of Volumes 2a and 2b as provided by Bell Laboratories.

General Works

39. Bug Fixes and changes in 4.2BSD.
    A brief discussion of the major user-visible changes made to the system since the last release.

Getting Started

40. An introduction to the C shell
    Introducing a popular command interpreter and many of the commonly used commands, assuming little prior knowledge of UNIX.
41. An Introduction to Display Editing with Vi
    The document to read to learn to use the *vi* screen editor.
42. Edit: A tutorial (Revised)
    For those who prefer line oriented editing, an introduction assuming no previous knowledge of UNIX or of text editing.
43. Ex Reference Manual (Version 3.1 — Oct. 1980)
    The final reference for the *ex* editor, which underlies both *edit* and *vi.*
44. Ex Changes — Version 3.1 to 3.5
    A quick guide to what is new in version 3.5 of *ex* and *vi,* for those who have used version 3.1.
45. Mail Reference Manual (Revised)
    Complete details on the mail processing program.
46. A Guide to the Dungeons of Doom (Revised)
    An introduction to the popular game of *rogue.*

Languages

47. The FRANZ LISP Manual
    A dialect of LISP, largely compatible with MACLISP.

† UNIX is a trademark of Bell Laboratories.

48. Berkeley Pascal User's Manual
    An interpretive implementation of the reference language.

49. The Programming Language EFL
    An introduction to a powerful FORTRAN preprocessor providing access to a language with structures much like C.

50. Berkeley FP User's Manual
    A description of the Berkeley implementation of Backus' Functional Programming Language, FP.

51. A Portable Fortran 77 Compiler
    A revised version of the document which originally appeared in Volume 2b; this version reflects the ongoing work at Berkeley.

52. Introduction to the f77 I/O Library
    A description of the revised input/output library for Fortran 77. This document, which originally appeared in Volume 2b, reflects the work carried out at Berkeley.

Document preparation

53. Writing Papers with *nroff* using −me
    A popular macro package for *nroff*.

54. −me Reference Manual
    The final word on −me.

55. The Berkeley Font Catalog
    Samples of fonts currently available for the raster plotters.

56. Writing tools − the Style and Diction Programs
    Description of programs which help you understand and improve your writing style.

57. Refer − A Bibliography System
    An introduction to the tools used to maintain bibliographic databases. The major program, *refer*, is used to automatically retrieve and format references based on document citations.

58. A Revised Version of −ms
    A quick description of the revisions made to the −ms formatting macros for nroff and troff.

Programming

59. Assembler Reference Manual
    For compiler writers.

✓ 60. Screen Updating and Cursor Movement Optimization
    An aide for writing screen-oriented, terminal independant programs.

61. An Introduction to the Source Code Control System
    A useful introductory article for those users who are licensed for SCCS.

System Installation and Administration

62. Installing and Operating 4.2BSD on the VAX
    The definitive reference document for those occasions when you find you need to start over again.

63. Building 4.2BSD UNIX Systems with Config
    An in-depth discussion of the use and operation of the *config* program. This document discusses how to configure and build binary images of UNIX for your site.

64. Disc Quotas in a UNIX Environment
    A light introduction to the care and feeding of the facilities which can be used in limiting disc resources.

65. 4.2BSD Line Printer Spooler Manual
    This document describes the structure and installation procedure for the line printer spooling system.

66. Fsck — The UNIX File System Check Program
    A reference document for use with the *fsck* program during times of file system distress.

67. Sendmail Installation and Operation Guide
    The last word in installing and operating the *sendmail* program.

Supporting Documentation

68. 4.2BSD System Manual
    A concise, though terse, description of the system call interface provided in 4.2BSD. This will never be a best seller.

69. A Fast File System for UNIX
    A description of the new file system organization design and implementation.

70. 4.2BSD Networking Implementation Notes
    A concise description of the system interfaces used within the networking subsystem.

71. Sendmail — An Internetwork Mail Router
    An overview document on the design and implementation of *sendmail*.

Bug fixes and changes in 4.2BSD

July 28, 1983

Samuel J. Leffler

Computer Systems Research Group
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720
(415) 642-7780

ABSTRACT

This document describes briefly the changes in the Berkeley system for the VAX between the 4.1BSD distribution of April 1981 and this, its revision of July 1983. It attempts to summarize, without going into great detail, the changes which have been made.

Notable improvements

- The file system organization has been redesigned to provide at least an order of magnitude improvement in disk bandwidth.
- The system now provides full support for the DOD Standard TCP/IP network communication protocols. This support has been integrated into the system in a manner which allows the development and concurrent use of other communication protocols. Hardware support and routing have been isolated from the protocols to allow sharing between varying network architectures. Software support is provided for 10 different hardware devices including 3 different 10 Mb/s Ethernet modules.
- A new set of interprocess communication facilities has replaced the old multiplexed file mechanism. These new facilities allow unrelated processes to exchange messages in either a connection-oriented or connection-less manner. The interprocess communication facilities have been integrated with the networking facilities (described above) to provide a single user interface which may be used in constructing applications which operate on one or more machines.
- A new signal package which closely models the hardware interrupt facilities found on the VAX replaces the old signals and jobs library of 4.1BSD. The new signal package provides for automatic masking of signals, sophisticated signal stack management, and reliable protection of critical regions.
- File names are now almost arbitrary length (up to 255 characters) and a new file type, symbolic link, has been added. Symbolic links provide a "symbolic referencing" mechanism similar to that found in Multics. They are interpolated during pathname expansion and allow users to create links to files and directories which span file systems.
- The system supports advisory locking on files. Files can have "shared" or "exclusive" locks applied by processes. Multiple processes may apply shared locks, but only one process at any time may have an exclusive lock on a file. Further, when an exclusive lock is present on a file, shared locks are disallowed. Locking requests normally block a process

until they can be completed, or they may be indicated as "non-blocking" in which case an error is returned if the lock can not be immediately obtained.

- The group identifier notion has been extended to a "group set". When users log in to the system they are placed in all their groups. Access control is now done based on the group set rather than just a single group id. This has obviated the need for the newgrp command.
- Per-user, per-filesystem disk quotas are now part of the system. Soft and hard limits may be specified on a per user and per filesystem basis to control the number of files and amount of disk space allocated to a user. Users who exceed a soft limit are warned and if, after three login sessions, their disk usage has not dropped below the soft limit, their soft limit is treated as a hard limit. Utilities exist for the creation, maintenance, and reporting of disk quotas.
- System time is now available in microsecond precision and millisecond accuracy. Users are provided with 3 high-resolution timers which may be set up to automatically reload on expiration. The timers operate in real time, user time, and process virtual time (for profiling). All statistics and times returned to users are now given in a standard format with seconds and microseconds separated. This eliminates program dependence on the line clock frequency.
- A new system call to rename files in the same file system has been added. This call eliminates many of the anomalies which could occur in older versions of the system due to lack of atomicity in removing and renaming files.
- A new system call to truncate files to a specific length has been added. This call improves the performance of the Fortran I/O library.
- Swap space configuration has been improved by allowing multiple swap partition of varying sizes to be interleaved. These partitions are sized at boot time minimize configuration dependencies.
- The Fortran 77 compiler and associated I/O library have undergone extensive changes to improve reliability and performance. Compilation may, optionally, include optimization phases to improve code density and decrease execution time.
- A new symbolic debugger, dbx, replaces the old symbolic debugger sdb. Dbx works on both C and Fortran 77 programs and allows users to set break points and trace execution by source code line numbers, references to memory locations, procedure entry, etc. Dbx allows users to reference structured and local variables using the program's programming language syntax.
- The delivermail program has been replaced by sendmail. Sendmail provides full internetwork routing, domain style naming as defined in the DARPA Request For Comments document #833, and eliminates the compiled in configuration database previously used by delivermail. Further, sendmail uses the DARPA standard Simple Mail Transfer Protocol (SMTP) for mail delivery.
- The system contains a new line printer system. Multiple line printers and spooling queues are supported through a printer database file. Printers on serial lines, raster printing devices, and laser printers are supported through a series of filter programs which interface to the standard line printer "core programs". A line printer control program, lpc, allows printers and printer queues to be manipulated. Spooling to remote printers is supported in a transparent fashion.
- Cu has been replaced by a new program tip. Tip supports a number of auto-call units and allows destination sites to be specified by name rather than phone number. Tip also supports file transfer to non-UNIX machines and can be used with sites which require half-duplex and/or odd-even parity.
- Uucp now supports many auto-call units other than the DN11. Spooling has been reorganized into multiple directories to cut down on system overhead. Several new utilities

and shell scripts exist for use in adminstrating uucp traffic. Operation has been greatly improved by numerous bug fixes.

Bug fixes and changes

Section 1

adb Support has been added for interpreting kernel data structures on a running system and in post mortem crash dumps created by savecore. A **−k** option causes adb to map addresses according to the system and current process page tables. A new command, $p, can be used to switch between process contexts. Many scripts are available for symbolically displaying kernel data structures, searching for a process' context by process ID, etc. A new document, "Using ADB to Debug the UNIX Kernel", supplies hints in the use of adb with system crash dumps.

addbib Is a new utility for creating and extending bibliographic data bases for use with refer.

apply Is a new program which may be used to apply a command to a set of arguments.

ar Has a new key, 'o', for preserving a file's modification time when it is extracted from an archive.

cc Supports the additional symbol information used by dbx. The old symbol information, used by the defunct sdb debugger, is available by specifying the **−go** flag. A new flag, **−pg**, creates executable programs which collect profiling information to be interpreted by the new gprof program. A bug in the C preprocessor, which caused line numbers to be incorrect for macros with formal parameters with embedded newlines has been fixed. The C preprocessor now properly handles hexadecimal constants in "#if" constructs and checks for missing "#endif" statements.

chfn Now works interactively in changing a user's information field in the password file.

chgrp Is now in section 1 and may be executed by anyone. Users other than the super-user may change group ownership of a file they own to any group in their group access list.

cp Now has a **−r** flag to copy recursively down a file system tree.

csh A bug which caused backquoted commands to wedge the terminal when interrupted has been fixed. Job identifiers are now globbed. A bug which caused the "wait" command to uninteruptible in certain cases has been fixed. History may now be saved and restored across terminal sessions with the *savehist* variable. The newgrp command has been deleted due to the new group facilities.

ctags Now handles C **typedefs**.

cu Exists only in the form of a "compatible front-end" to the new tip program.

dbx Is a new symbolic debugger replacing sdb. Dbx handles C and Fortran programs.

delivermail
Has been replaced by the new sendmail program.

df Understands the new file system organization and reports all disk space totals in kilobytes.

du Now reports disk usage in kilobytes and uses the new field in the inode structure which contains the actual number of blocks allocated to a file to increase accuracy of calculations.

dump Has been moved to section 8.

error Has been taught about the error message formats of troff.

eyacc A bug which caused the generated parser to not recognize valid null statements has been fixed.

f77 Has undergone major changes.

The i/o library has been extensively tested and debugged. Sequential files are now opened at the BEGINNING by default; previously they were opened at the end.

Compilation of data statements has been substantially sped up. Significant new optimization is optionally available (this is still a bit buggy and should be used with caution). Even without optimization, however, single precision computations execute much faster.

The new debugger, dbx, has replaced sdb for debugging Fortran programs; sdb is no longer supported.

Files with ".F" suffixes are preprocessed by the C preprocessor. This allows C-style "#include" and "#define" constructs to be used. The compiler has been modified to print error messages with sensible line numbers. Make also understands the ".F" suffix. Note that when using the C preprocessor, the 72 column convention is not followed.

The **−I** option for specifying short integers has been changed to **−i**. The **−I** option is now used to specify directory search paths for "#include" statements. A **−pg** option for creating executable images which collect profiling information for gprof has been added.

fed Is a font editor of dubious value.

file Now understands symbolic links.

find Has a new **−type** value, 'l', for finding symbolic links.

fp Is a new compiler/interpreter for the Functional Programming language. A supporting document is present in Volume 2C of the UNIX Programmer's Manual.

fpr Is a new program for printing Fortran files with embedded Fortran carriage controls.

fsplit Is a new program for splitting a multi-function Fortran file into individual files.

ftp Is a new program which supports the ARPA standard File Transfer Protocol.

gcore Is a new program which creates a core dump of a running process.

gprof Is a new profiling tool which displays execution time for the dynamic call graph of a program. Gprof works on C, Fortran, and Pascal programs compiled with the **−pg** option. Gprof may also be used in creating a call graph profile for the operating system. A supporting document, "gprof: A Call Graph Execution Profiler" is included in Volume 2C of the UNIX Programmer's Manual.

groups Is a new program which displays a user's group access list.

hostid Is a new program which displays the system's unique identifier as returned by the new gethostid system call. The super-user uses this program to set the host identifier at boot time.

hostname Is a new program which displays the system's name as returned by the new gethostname system call. The super-user uses this program to set the host name at boot time.

indent Is a new program for formatting C program source.

install Is a shell script used in installing software.

iostat Now reports kilobytes per second transferred for each disk. This is useful as the unit of information transferred is no longer a constant one kilobytes.

last Now displays the remote host from which a user logged in (when accessing a machine across a network). The pseudo user "ftp" may be specified to find out information about FTP file transfer sessions.

lastcomm Now displays flags for each command indicating if the program dumped core, used PDP-11 mode, executed with a set-user-ID, or was created as the result of a fork (with no following exec).

learn Now has lessons for vi (this is user contributed software which is not part of the standard system).

lint Has a new **−C** flag for creating lint libraries from C source code. Has improved type checking on static variables.

lisp Has been ported to several 68000 UNIX systems, the relevant code is included in the distribution. A new vector data type and a form of "closure" have been added.

ln Has a new flag, **−s**, for creating symbolic links.

login Has been extensively modified for use with the rlogind and telnetd network servers.

lpq Is totally new, see lpr.

lpr And its related programs are totally new. The line printer system supports multiple printers of many different characteristics. A master data base, /etc/printcap, describes both local printers and printers accessable across a network. A document describing the line printer system is now part of Volume 2C of the UNIX Programmer's Manual.

lprm Is totally new, see lpr.

ls Has been rewritten for the new directory format. It understands symbolic links and uses the new inode field which contains the actual number of blocks allocated to a file when the **−s** flag is supplied. Many rarely used options have been deleted.

m4 A bug which caused m4 to dump core when keywords were undefined then redefined has been fixed.

Mail Now supports mail folders in the style of the Rand MH system. Has been reworked to cooperate with sendmail in understanding the new mail address formats. Allows users to defined message header fields which are not be displayed when a messages is viewed. Many other changes are described in a revised version of the user manual.

make Understands not to unlink directories when interrupted. Understands the new ".F" suffix for Fortran source files which are to be run through the C preprocessor. Has a new predefined macro MFLAGS which contains the flags supplied to make on the command line (useful in creating hierarchies of makefiles).

mkdir Now uses the mkdir system call to run faster.

mt Has a new command, status, which shows the current state of a tape drive.

mv Has been rewritten to use the new rename system call. As a result, multiple directories may now be moved in a single command, the restrictions on having ".." in a pathname are no longer present, and everything runs faster.

net And all related Berknet programs are no longer part of the standard distribution. These programs live on in /usr/src/old for those who can not do without them.

netstat Is a new program which displays network statistics and active connections.

oldcsh No longer exists.

od Has gobs of new formats options.

pagesize Is a new program which prints the system page size for use in constructing portable shell scripts.

passwd Now reliably interlocks with chsh, chfn, and vipw, in guarding against concurrent updates to the password file.

pc/pi **For** loops are now done according to the standard. Files may now be dynamically allocated and disposed. Records and variant records are now aligned to correspond to C structures and unions (this was falsely claimed before). Several obscure bugs involving formal routines have been fixed. Three new library routines support random access file i/o, see /usr/include/pascal for details.

pc **For** loop variables and **with** pointers are now allocated to registers. Separate compilation type checking can now be done without reference to the source file; this permits movement (including distribution) of .o files and creation of libraries. Display entries are saved only when needed (a speed optimization).

pdx Is a new debugger for use with pi. Pdx is invoked automatically by the interpreter if a run-time error is encountered. Future work is planned to extend the new dbx debugger to understand code generated by the Pascal compiler pc.

ps Has been changed to work with the new kernel and is no longer dependent on system page size. All process segment sizes are now shown in kilobytes. Understands that the old "using new signal facilities" bit in the process structure now means "using old 4.1BSD signal facilities".

pwd Now simply calls the *getwd*(3) routine.

rcp Is a new program for copying files across a network. The complete syntax of cp is supported, including recursive directory copying.

refer Has had many bugs fixed in it and the associated −ms macro package support made to work.

reset Now resets all the special characters to the system defaults specified in the include file <sys/ttychars.h>.

rlogin Is a new program for logging in to a machine across a network. Rlogin uses the files /etc/hosts.equiv and .rhosts in the users login directory to allow logins to be performed without a password. Rlogin supports proper handling of ^S/^Q and flusing of output when an interrupt is typed at the terminal. Its '~' escape sequences are reminiscent of the old cu program (as it is based on the same source code).

rmdir Now uses the rmdir system call to run more efficiently and not require root privileges. Unfortunately, this means arguments which end in one or more "/" characters are no longer legal.

roffbib Is a new program for running off bibiliographic databases.

rsh Is a new program which supports remote command execution across a network.

ruptime Is a new program which displays system status information for clusters of machines attached to a local area network.

rwho Is a new program which displays users logged in on clusters of machines attached to a local area network.

script Has been rewritten to use pseudo-terminals. This allows the C shell job control facilities (among other things) to be used while scripting. A side effect of this change is that scripts now contain everything typed a terminal.

sdb Has been replaced by dbx; it still lives on in /usr/src/old for those with a personal attachment.

sendbug Is a new command for submitting bug reports on 4.2BSD in a standard format suitable for automatic filing by the bugfiler program.

sh No longer has a newgrp command due to the new groups facilities.

sortbib Is a new command for sorting bibliographic databases.

strip Has been made blindingly fast by using the new truncate system call (thereby eliminating the old method of copying the file).

stty The default system erase, kill, and interrupt characters have been made the DEC standard values of DEL ('^?'), '^U', and '^C'. This is not expected to gain much popularity, but was done in the interest of compatibility with many other standard operating systems.

su Has been changed to do a "full login" when starting up the subshell. A new flag, **−f**, does a "fast" su for when a system is heavily loaded. Extra arguments supplied to su are now treated as a command line and executed directly instead of creating an interactive shell.

sysline Is a new program for maintaining system status information on terminals which support a "status line"; a poor man's alternative to a window manager (or emacs).

tail Has a larger buffer so that "tail −r" and similar show more.

talk Is a new program which provides a screen-oriented write facility. Users may be "talked to"across a network, though satellite response times have indicated overseas conversations are still best done by phone. Can be very obnoxious when engaged in important work.

tar Now allocates its internal buffers dynamically so that the block size can be specified to be very large for streaming tape drives. Also, now avoids many core-core copy operations. Has a new **−C** option for forcing chdir operations in the middle of operation (thereby allowing multiple disjoint subtrees to be easily placed in a single file, each with short relative pathnames). Has a new flag, 'B', for forcing 20 block records to be read and written; useful in joining two tar commands with a remote shell to transfer large amounts of data across a network.

telnet Is a new program which supports the ARPA standard Telnet protocol.

tip Replaces cu as the standard mechanism for connecting to machines across a phone line or through a hardwired connection. Tip uses a database of system descriptions, supports many different auto-call units, and understands many nuances required to talk to non-UNIX systems. Files may be transferred to and from non-UNIX systems in a simple fashion.

ul A bug which sometimes caused an extra blank line to be printed after reaching end of file has been fixed.

uucp And related programs have been extensively enhanced to support many different auto-call units and multiple spooling directories (among other things). A large number of bugs and performance enhancements have been made.

uusnap Is a new program which gives a snap-shot of the uucp spooling area.

vfontinfo Is a program used to inspect and print information about fonts.

vgrind Now uses a regular expression language to describe formatting. A **−f** flag forces vgrind to act as a filter, generating output suitable for inclusion in troff and/or nroff documents. Language descriptions exist for C, Pascal, Model, C shell, Bourne shell, Ratfor, and Icon programs.

vi A bug which caused the ^B command to place the cursor on the wrong line has been fixed. A bug which caused vi to believe a file had been modified when an i/o error occurred has been fixed. A bug which allowed "hardtabs" to be set to 0 causing a divide by zero fault has been fixed.

vlp Is a new program for pretty printing Lisp programs.

vmstat Has had one new piece of information added to **−s** summary, the number of fast page reclaims performed. The fields related to paging activity are now all given in kilobytes.

vpr And associated programs for spooling and printing files on Varian and Versatec printers are now shell scripts which use the new line printer support.

vwidth Is a new program for making troff width tables for a font.

wc Is once again identical to the version 7 program. That is, the **−v**, **−t**, **−b**, **−s**, and **−u** flags have been deleted.

whereis Understands the new directory organization for the source code.

which Now understands how to handle aliases.

who Now displays the remote machine from which a user is logged in.

Section 2.

The most important change in section 2 is that the documentation has been significantly improved. Manual page entries now indicate the possible error codes which may be returned and how to interpret them. The introduction to section 2 now includes a glossary of terms used throughout the section. The terminology and formatting have been made consistent. Many manual pages now have "NOTES" or "CAVEATS" providing useful information heretofore left out for the sake of brevity. As always the manual pages are still for the programmer; they are terse and extremely concise. The "4.2BSD System Manual" is likewise concise, but a bit more verbose in providing an overall picture of the system facilities.

With regard to changes in the facilities, these fall into three major categories: interprocess communication, signals, and file system related calls. The interprocess communication facilities center around the *socket* mechanism described in the "A 4.2BSD Interprocess Communication Primer". The new signals do not have an accompanying document, so the manual pages should be studied carefully. The new file system calls pretty much stand on their own, with a late section of the document "A Fast File System for UNIX" supplying a quick overview of the most important new file system facilities. Finally, it should be noted that the job control facilities introduced in 4.1BSD have been adopted as a standard part of 4.2BSD. No special distinction is given to these calls (in 4.1BSD they were earmarked "2J").

Many of the new system calls have both a "set" and a "get" form. Only the "get" forms are indicated below. Consult the manual for details on the "set" form.

intro Has been updated to reflect the new list of possible error codes. Now includes a glossary of terminology used in section 2.

access Now has symbolic definitions for the *mode* parameter defined in <*sys/file.h*>.

bind Is a new interprocess communication system call for binding names to sockets.

connect Is a new interprocess communication system call for establishing a connection between two sockets.

creat Has been obsoleted by the new *open* interface.

fchmod Is a new system call which does a *chmod* operation given a file descriptor; useful in conjunction with the new advisory locking facilities.

fchown Is a new system call which does a *chown* operation given a file descriptor; useful in conjunction with the new advisory locking facilities.

fcntl Is a new system call which is useful in controlling how i/o is performed on a file descriptor (non-blocking i/o, signal drive i/o). This interface is compatible with the System III fcntl interface.

flock Is a new system call for manipulating advisory locks on files. Locks may be shared or exclusive and locking operations may be indicated as being non-blocking, in which case a process is not blocked if the requested lock is currently in use.

fstat Now returns a larger stat buffer; see below under stat.

fsync Is a new system call for synchronizing a file's in-core state with that on disk. Its intended use is in building transaction oriented facilities.

ftruncate Is a new system call which does a *truncate* operation given a file descriptor; useful in conjunction with the new advisory locking facilities.

getdtablesize Is a new system call which returns the size of the descriptor table.

getgroups Is a new system call which returns the group access list for the caller.

gethostid Is a new system call which returns the unique (hopefully) identifier for the current host.

gethostname Is a new system call which returns the name of the current host.

getitimer Is a new system call which gets the current value for an interval timer.

getpagesize Is a new system call which returns the system page size.

getpriority Is a new system call which returns the current scheduling priority for a specific process, a group of processes, or all processes owned by a user. In the latter two cases, the priority returned is the highest (lowest numerical value) enjoyed by any of the specified processes.

getrlimit Is a new system call which returns information about a resource limit. The getrlimit and setrlimit calls replace the old vlimit call from 4.1BSD.

getrusage Is a new system call which returns information about resource utilitization of a child process or the caller. This call replaces the vtimes call of 4.1BSD.

getsockopt Is a new interprocess communication system call which returns the current options present on a socket.

gettimeofday Is a new system call which returns the current Greenwich date and time, and the current timezone in which the machine is operating. Time is returned in seconds and microseconds since January 1, 1970.

ioctl Has been changed to encode the size of parameters and whether they are to be copied in, out, or in and out of the user address space in the *request.* The symbolic names for the various ioctl requests remain the same, only the numeric values have changed. A number of new ioctls exist for use with sockets and the network facilities. The old LINTRUP request has been replaced by a call to fcntl and the SIGIO signal.

killpg Has now been made a system call; in 4.1BSD it was a library routine.

listen Is a new interprocess communication system call used to indicate a socket will be used to listen for incoming connection requests.

lseek Now has symbolic definitions for its *whence* parameter defined in <*sys/file.h*>.

mkdir Is a new system call which creates a directory.

mpx The multiplexed file facilities are no longer part of the system. They have been replaced by the socket, and related, system calls.

open Is different, now taking an optional third parameter and supporting file creation, automatic truncation, automatic append on write, and "exclusive" opens. The open interface has been made compatible with System III with the exception that non-blocking opens on terminal lines requiring carrier are not supported.

profil Now returns statistical information based on a 100 hz clock rate.

quota Is a new system call which is part of the disk quota facilities. Quota is used to manipulate disk quotas for a specific user, as well as perform certain random chores such as syncing quotas to disk.

read Now automatically restarts when a read on a terminal is interrupted by a signal before any data is read.

readv Is a new system call which supports scattering of read data into (possibly) disjoint areas of memory.

readlink Is a new system call for reading the value of a symbolic link.

recv Is a new interprocess communication system call used to receive a message on a connected socket.

recvfrom Is a new interprocess communication system call used to receive a message on a (possibly) unconnected socket.

recvmsg Is a new interprocess communication system call used to receive a message on a (possibly) unconnected socket which may have access rights included. When using on-machine communication, recvmsg and sendmsg may be used to pass file descriptors between processes.

rename Is a new system call which changes the name of an entry in the file system (plain file, directory, character special file, etc.). Rename has an important property in that it guarantees the target will always exist, even if the system crashes in the middle of the operation. Rename only works with source and destination in the same file system.

rmdir Is a new system call for removing a directory.

select Is a new system call (mainly for interprocess communication) which provides facility for synchronous i/o multiplexing. Sets of file descriptors may be queried for readability, writability, and if any exceptional conditions are present (such as out of band data on a socket). An optional timeout may also be supplied in which case the select operation will return after a specified period of time should no descriptor satisfy the requests.

send Is a new interprocess communication system call for sending a message on a connected socket.

sendto Is a new interprocess communication system call for sending a message on a (possibly) unconnected socket.

sendmsg Is a new interprocess communication system call for sending a message on a (possibly) unconnected socket which may included access rights.

setquota Is a new system call for enabling or disabling disk quotas on a file system.

setregid Is a new system call which replaces the 4.1BSD setgid system call. Setregid allows the real and effective group ID's of a process to be set separately.

setreuid Is a new system call which replaces the 4.1BSD setuid system call. Setreuid allows the real and effective user ID's of a process to be set separately.

shutdown Is a new interprocess communication system call for shutting down part or all of full-duplex connection.

sigblock Is a new system call for blocking signals during a critical section of code.

sigpause Is a new system call for blocking a set of signals and then pausing indefinitely for a signal to arrive.

sigsetmask Is a new system call for setting the set of signals which are currently blocked from delivery to a process.

sigstack Is a new system call for defining an alternate stack on which signals are to be processed.

sigsys Is no longer supported. The new signal facilities are a superset of those which sigsys provided.

sigvec Is the new system call interface for defining signal actions. For each signal (except SIGSTOP and SIGKILL), sigvec allows a "signal vector" to be defined. The signal vector is comprised of a handler, a mask of signals to be blocked while the handler executes, and an indication of whether or not the handler should execute on a

signal stack defined by a sigstack call. The old signal interface is provided as a library routine with several important caveats. First, signal actions are no longer reset to their default value after a signal is delivered to a process. Second, while a signal handler is executing the signal which is being processed is blocked until the handler returns. To simulate the old signal interface, the user must explicitly reset the signal action to be the default value and unblock the signal being processed.

Four new signals have been added for the interprocess communication and interval timer facilities. SIGIO is delivered to a process when an fcntl call enables signal driven i/o and input is present on a terminal (and a signal handler is defined). SIGURG is delivered when an urgent condition arises on a socket (and a handler is defined). SIGPROF and SIGVTALRM are associated with the ITIMER_PROF and ITIMER_VIRTUAL interval timers, and delivered to a process when such a timer expires (the SIGALRM signal is used for the ITIMER_REAL interval timer). The old SIGTINT signal is replaced by SIGIO.

socket Is a new interprocess communication system call for creating a socket.

socketpair Is a new interprocess communication system call for creating a pair of connected and unnamed sockets.

stat Now returns a larger structure. New fields are present indicating the optimal blocking factor in which i/o should be performed (for disk files the block size of the underlying file system) and the actual number of disk blocks allocated to the file. Inode numbers are now 32-bit quantities. Several spare fields have been allocated for future expansion. These include space for 64-bit file sizes and 64-bit time stamps. Two new file types may be returned, S_IFLNK for symbolic links, and S_IFSOCK for sockets residing in the file system.

swapon Has been renamed from the vswapon call of 4.1BSD.

symlink Is a new system call for creating a symbolic link.

truncate Is a new system call for truncating a file to a specific size.

unlink Should no longer be used for removing directories. Directories should only be created with mkdir and removed with rmdir. Creating hard links to directories can cause disastrous results.

utime Is defunct, replaced by utimes.

utimes Is a new system call which uses the new time format in setting the accessed and updated times on a file.

vfork Is still present, but definitely on its way out. Future plans include implementing fork with a scheme in which pages are initially shared read-only. On the first attempt by a process to write on a page the parent and child would receive separate writable copies of the page.

vlimit Is no longer supported. Vlimit is replaced by the getrlimit and setrlimit calls.

vread Is no longer supported in the system.

vswapon Has been renamed swapon.

vtimes Is no longer supported. Vtimes is replaced by the getrusage call.

vwrite Is no longer supported in the system.

wait Now is automatically restarted when interrupted by a signal before status could be returned.

wait3 Returns resource usage in a different format than that which was returned in 4.1BSD. This structure is compatible with the new getrusage system call. Wait3 is now automatically restarted when interrupted by a signal before status could be returned.

write Now is automatically restarted when writing on a terminal and interrupted by a signal before any i/o was completed.

writev Is a new version of the write system call which supports gathering of data in (possibly) discontiguous areas of memory

Section 3

The section 3 documentation has been reorganized to group manual entries by library. Introductory sections for each logical and physical library contain lists of the entry points in the library.

A number of routines which had been system calls under 4.1BSD are now user-level library routines in 4.2BSD. These routines have been grouped under section "3C" headings, "C" for compatibility. Further, certain routines present in the standard C run-time library which do not easily categorize as part of one of the standard libraries, have been group under "3X" headings.

curses A number of bug fixes have been incorporated, and the documentation has been revised.

stdio The standard i/o library has been modified to block i/o operations to disk files according to the block size of the underlying file system. This is accomplished using the new *st_blksize* value returned by *fstat.* The resultant performance improvement is significant as the old 1 kilobyte buffer size often resulted in 7 memory-to-memory copy operations by the system on 8 kilobyte block file systems.

End-of-file marks now "stick". That is, all input requests on a stdio channel after encountering end-of-file will return end-of-file until a *clearerr* call is made. This has implications for programs which use stdio to read from a terminal and do not process end-of-file as a terminating keystroke.

Two new functions may be used to control i/o buffering. The *setlinebuf* routine is used to change **stdout** or **stderr** from block buffered to unbuffered to line buffered. The *setbuffer* routine is an alternate form of *setbuf* which can be used after a stream has been opened, but before it is read or written.

bstring Three new routines, *bcmp*, *bcopy*, and *bzero* have been added to the library. These routines use the VAX string instructions to manipulate binary byte strings of a known size.

ctime Now uses the *gettimeofday* system call and supports time conversion in six different time zones. Daylight savings calculations are also performed in each time zone when appropriate.

isprint Now considers space a printing character; as the manual page has always indicated.

directory Is a new directory interface package which provides a portable interface to reading directories. A version of this library which operates under 4.1BSD is also available.

getpass Now properly handles being unable to open /dev/tty.

getwd Has been moved from the old jobs library to the standard C run-time library. It now returns an error string rather than printing on the standard error when unable to decipher the current working directory.

malloc The standard library malloc has NOT changed from 4.1BSD, because a new version which passed all our validation tests arrived too late. The newer malloc which resolves the problems the current malloc has with large virtual environment is located in /usr/src/local/malloc.c.

perror Now uses the writev system call to pass all its arguments to the system in a single system call. This has profound effects on programs which transmit error messages across a network.

psignal And sys_siglist are routines for printing signal names in an equivalent manner to perror.

qsort Has been greatly sped up by choosing a random element with which to apply its divide and conquer algorithm.

random Is a successor to rand which generates much better random numbers. The old rand routine is still available and most programs have not been switched over to random as doing so would make certain facilities such encrypted mail unable to operate on existing data files.

setjmp And longjmp now save and restore the signal mask so that non-local exit from a signal handler is transparent. The old semantics are available with _setjmp and _longjmp.

net Is a new set of routines for accessing database files for the DARPA Internet. Four databases exist: one for host names, one for network names, one for protocol numbers, and one for network services. The latter returns an Internet port and protocol to be used in accessing a given network service.

An additional collection of routines, all prefaced with "inet_" may be used to manipulate Internet addresses, and interpret and convert between Internet addresses and ASCII representations in the Internet standard "dot" notation.

Finally, routines are available for converting 16 and 32 bit quantities between host and network order (on high-ender machines these routines are defined to be noops).

fstab The routines for manipulating /etc/fstab have been rewritten to return arbitrary length null-terminated strings.

Section 4

The system now supports the 11/730, the new 64Kbit RAM memory controllers for the 11/750 and 11/780, and the second UNIBUS adapter for the 11/750. Several new character and/or block device drivers have been added, as well as support for many hardware devices which are accessible only through the network facilities. Each new piece of hardware supported is listed below.

New manual entries in section 4 have been created to describe communications protocols, and network architectures supported. At present the only network architecture fully supported is the DARPA Internet with the TCP, IP, UDP, and ICMP protocols.

acc A network driver for the ACC LH/DH IMP interface.

ad A driver for the Data Translation A/D converter.

arp The Address Resolution Protocol for dynamically mapping betwee 32-bit DARPA Internet addresses and 48-bit Xerox 10Mb/s Ethernet addresses.

css A network driver for the DEC IMP-11A LH/DH IMP interface.

dmc A network interface driver for the DEC DMC-11/DMR-11 point-to-point communications device.

ec A network interface driver for the 3Com 10Mb/s Ethernet controller.

en A network interface driver for the Xerox 3Mb/s experimental Ethernet controller.

hy A network interface driver for the Network Systems Hyperchannel Adapter.

ik A driver for an Ikonas frame buffer graphics device interface.

il A network interface driver for the Interlan 10Mb/s Ethernet interface.

imp A network interface driver for the standard 1822 interface to an IMP; used in conjunction with either acc or css hardware.

kg A driver for a KL-11/DL-11W used as an alternate real time clock source for gathering kernel statistics and profiling information.

lo A software loopback network interface for protocol testing and performance analysis.

pcl A network interface driver for the DEC PCL-11B communications controller.

ps A driver for an Evans and Sutherland Picture System 2 graphics device connected with a DMA interface.

pty Now includes a simple packet protocol to support flow controlled operation with mechanisms for flushing data to be read and/or written.

rx A driver for the DEC dual RX02 floppy disk unit.

ts Now supports TU80 tape drives.

tu The VAX-11/750 console cassette interface has been made somewhat usable when operating in single-user mode. The device driver employs assembly language pseudo-dma code for the reception of incoming packets from the cassette.

uda Now supports RA81, RA80, and RA60 disk drives.

un A network interface driver for an Ungermann-Bass network interface unit connected to the host via a DR-11W.

up Now supports ECC correction and bad sector handling. Also has improved logic for recognizing many different kinds of disk drives automatically at boot time.

uu A driver for DEC dual TU58 tape cartridges connected via a DL-11W interface.

va The Varian driver has been rewritten so that it may coexist on the same UNIBUS with devices which require exclusive use of the bus; i.e. RK07's.

vv A network interface driver for the Proteon proNET 10Mb/s ring network controller.

Section 5

dir Reflects the new directory format.

disktab Is a new file for maintaining disk geometry information. This is a temporary scheme until the information stored in this file for each disk is recorded on the disk pack itself.

dump Is a superset of that used in 4.1BSD.

fs Reflects the new file system organization.

gettytab Is a new file which idescribes terminal characteristics. Each entry in the file describes one of the possible arguments to the getty program.

hosts Is a database for mapping between host names and DARPA Internet host addresses.

mtab Has been modified to include a "type" field indicating whether the file system is mounted read-only, read-write, or read-write with disk quotas enabled.

networks Is a database for mapping between network names and DARPA standard network numbers.

phones Is a phone number data base for tip.

printcap Is a termcap clone for configuring printers.

protocols Is a database for mapping between protocol names and DARPA Internetwork standard protocol numbers.

remote Is a database of remote hosts for use with tip.

tar Is a new entry describing the format of a tar tape.

utmp Has been augmented to include a remote host from which a login session originates. The wtmp file is also used to record FTP sessions.

vgrindefs Is a file describing how to vgrind programs written in many languages.

Section 6

aardvark Does not work because it requires the "Dungeon Definition Language" processor which is a binary image requiring 4.1BSD compatibility mode; the DDL source is still present.

aliens The aliens have returned home, the game is no longer included in the distribution.

backgammon
Is now screen oriented. A new program, teachgammon, instructs the new backgammon player. The old version is now called btlgammon.

canfield Is a new game which plays a brand of the popular game of solitaire. Betting is included, the program cfscores may be used to find out your current debt.

ching Now pipes its output through more. Thus the hacker placates the seekers.

chase No longer exists because the binary does not work under 4.2BSD.

factor Is a rewrite in C of the old version 7 assembly language program which finds the prime factors of a number.

fortune Has yet more adages.

hangman Is now screen oriented.

mille Now plays more intelligently.

primes Is a rewrite in C of the old version 7 assembly language program which finds prime numbers within a specified range.

rogue Has been made more of a scoundrel. The supplementary document "A Guide to the Dungeons of Doom", has been updated as well, and is now part of Volume 2C of the programmer's manual.

sail Is a new game which simulates sea battles of yore. The manual page is large enough to be a separate document and so has been left in its source directory.

trek The original trek has returned; trekies rejoice.

Section 7

hier Has been updated to reflect the reorganization to the user and system source.

mailaddr Is a new entry describing mail addressing syntax under sendmail (possibly too Berkeley specific).

ms The −ms macros have been extended to allow automatic creation of a table of contents. Support for the refer preprocessor is improved. Several bugs related to multi-column output and floating keeps have been fixed. Extensions to the accent mark string set are available by including the .AM macro. Footnotes can now be automatically numbered (in superscript) by −ms and referenced in the text with a ** string register. The manual page includes a summary of important number and string registers. A new document "Changes to −ms" is included in Volume 2C of the programmer's manual.

Section 8

Major changes affecting system operations include:

- The system now supports disk quotas. These allow system administrators to control users' disk space and file allocation on a per-file system basis. Utilities in this section exist for fixing, summarizing, and editing disk quota summary files.
- File systems are now made with a new program, newfs, which acts as front end to the old mkfs program. There no longer is a need to remember disk partition sizes, as newfs gets this information automatically from the /etc/disktab file. In addition, newfs attempts to lay out file systems according to the characteristics of the underlying disk drive (taking into account disk geometry information).
- DEC standard bad block forwarding is now supported on the RP06 and second vendor UNIBUS storage module disks. The bad144 program can now be used to mark sectors bad on many disks, though inclusion in the bad sector table is still somewhat risky due to requirements in the ordering of entries in the table.
- A new program, format, should be used to initialize all non-DEC storage modules before creating file systems. Format formats the sector headers and creates a bad sector table which is used in normal system operation. Format runs in a standalone mode.
- Getty has been rewritten to use a description file, /etc/gettytab. This allows sites to tailor terminal operation and configuration without making modifications to getty.
- The line printer system is totally new. A program to administer the operation of printers, lpc, is supplied, and printer accounting has been consolidated into a single program, pac.
- The program used to restore files from dump tapes is now called *restore*. This was done to reinforce the fact that it is completely rewritten and operates in a very different way than the old restor program. Restore operates on mounted file systems and uses only normal file system operations to restore files. Versions of both dump and restore which operate across a network are included as rdump and rrestore. Dump and restore (and their network oriented counterparts) now perform so efficiently (mostly because of the new file system), that disk to disk backups should no longer be an attractive alternative.

arff No longer asks if you want to clobber the floppy when manipulating archives which are not on the floppy.

bad144 Has been modified to use the /etc/disktab file. Can be used to create bad sector tables for the DEC RP06 and several new Winchester disk drives. Consult the source code for details and use with extreme care.

badsect Has been modified to work with the new file system and now must interact with fsck to perform its duties. Consult the manual page for more information.

bugfiler Is a new program for automatic filing and acknowledgement of bug reports submitted by the sendbug program. Intended to operate with the Rand MH software which is part of the user contributed software. Used at Berkeley to process bug reports on 4.2BSD.

chgrp Has been moved to section 1.

comsat Has been changed to filter the noise lines in message headers when displaying incoming mail. No longer uses a second process watchdog as it uses the more reliable socket facilities instead of the old mpx facilities.

config Has been extensively modified to handle the new root and swap device specification syntax. A new document, "Configuring 4.2BSD UNIX Systems with Config", describes its use, as well as other important information needed in configuring system images; this is part of Volume 2C of the programmer's manual.

diskpart Is a new program which may be used to generate disk partition tables according to the rules used at Berkeley. Can automatically generate entries required for device drivers and for the /etc/diskpart file. (Does not handle the new DEC DSA style drives properly because it tries to reserve space for the bad sector table.)

drtest Is a new standalone program which is useful in testing standalone disk device drivers and for pinpointing bad sectors on a disk.

dump Has been modified for the new file system organization. Mainly due to the new file system, it runs virtually at tape speed. Properly handles locking on the dumpdates file when multiple dumps are performed concurrently on the same machine.

dumpfs Is a new program for dumping out information about a file system such as the block size and disk layout information.

edquota Is a new program for editing user quotas. Operates by invoking your favorite editor on an ASCII representation of the information stored in the binary quota files. Edquota also has a "replication" mode whereby a quota template may be used to create quotas for a group of users.

fastboot Is a new shell script which reboots the system without checking the file systems; should be used with extreme care.

fasthalt Is a new script which is similar to fastboot.

format Is a new standalone program for formatting non-DEC storage modules and creating the appropriate bad sector table on the disk.

fsck Has been changed for the new file system. Fsck is more paranoid then ever in checking the disks, and has been sped up significantly. The accompanying Volume 2C document has been updated to reflect the new file system organization.

ftpd Is the DARPA File Transfer Protocol server program. It supports C shell style globbing of arguments and a large set of the commands in the specification (except the ABORT command!).

gettable Is a new program which can be used in aquiring up to date DARPA Internet host database files.

getty Has been rewritten to use a terminal description database, /etc/gettytab. Consult the manual entries for *getty*(8) and *gettytab*(5) for more information.

icheck Has been modified for the new file system.

init Has been significantly modified to use the new signal facilities. In doing so, several race conditions related to signal delivery have been fixed.

kgmon Is a new program for controlling running systems which have been created with kernel profiling. Using kgmon, profiling can be turned on or off and internal profiling buffers can dumped into a gmon.out file suiitable for interpretation by gprof.

lpc Is a new program controlling line printers and their associated spooling queues. Lpc can be used to enable and disable printers and/or their spooling queues. Lpc can also be used to rearrange existing jobs in a queue.

lpd Has been rewritten and now runs as a "server", using the interprocess communication facilities to service print requests. A supplementary document describing the line printer system is now part of Volume 2C of the programmer's manual.

MAKEDEV
Is a new shell script which resides in /dev and is used to create special files there. MAKEDEV keeps commands for creating and manipulating local devices in a separate file MAKEDEV.local.

mkfs Has been virtually rewritten for the new file system. The arguments supplied are very different. For the most part, users now use the newfs program when creating

file systems. Mkfs now automatically creates the lost+found directory.

mount Now indicates file systems which are mounted read-only or have disk quotas enabled.

newfs Is a new front-end to the mkfs program. Newfs figures out the appropriate parameters to supply to mkfs, invokes it, and then, if necessary, installs the boot blocks necessary to bootstrap UNIX on 11/750's.

pac Is a new program which can be used to do printer accounting on any printer. It subsumes the vpac program.

quot Now uses the information in the inode of each file to find out how many blocks are allocated to it.

quotacheck
Is a new program which performs consistency checks on disk quota files. Quotacheck is normally run from the /etc/rc.local file after a system is rebooted, though it can also be run on mounted on file systems which are not in use.

quotaon Is a new program which enables disk quotas on file systems. A link to quotaon, named quotaoff, is used to disable disk quotas on file systems.

pstat Has been modified to understand new kernel data structures.

rc Has had system dependent startup commands moved to /etc/rc.local.

rdump Is a new program to dump file systems across a network.

renice Has been rewritten to use the new setpriority system call. As a result, you can now renice users and process groups.

repquota Is a new program which summarizes disk quotas on one or more file systems.

restor No longer exists. A new program, restore, is its successor.

restore Replaces restor. Restore operates on mounted file systems; it contains an interactive mode and can be used to restore files by name. Restore has become almost as flexible to use as tar in retrieving files from tape.

rexecd Is a network server for the *rexec*(3X) library routine. Supports remote command execution where authentication is performed using user accounts and passwords.

rlogind Is a network server for the *rlogin*(1C) command. Supports remote login sessions where authentication is performed using privileged port numbers and two files, /etc/hosts.equiv and .rhosts (in each users home directory).

rmt Is a program used by rrestore and rdump for doing remote tape operations.

route Is a program for manually manipulating network routing tables.

routed Is a routing daemon which uses a variant of the Xerox Routing Information Protocol to automatically maintain up to date routing tables.

rrestore Is a version of restore which works across a network.

rshd Is a server for the *rsh*(1C) command. It supports remote command execution using privileged port numbers and the /etc/hosts.equiv and .rhosts files in users' home directories.

rwhod Is a server which generates and listens for host status information on local networks. The information stored by rwhod is used by the *rwho*(1C) and*ruptime*(1C) programs.

rxformat Is a program for formatting floppy disks (this uses the *rx* device driver, not the console floppy interface).

savecore Has been modified to get many pieces of information from the running system and crash dump to avoid compiled in constants.

sendmail Is a new program replacing delivermail; it provides fully internetwork mail forwarding capabilities. Sendmail uses the DARPA standard SMTP protocol to send and receive mail. Sendmail uses a configuration file to control its operation, eliminating the compiled in description used in delivermail.

setifaddr Is a new program used to set a network interface's address. Calls to this program are normally placed in the /etc/rc.local file to configure the network hardware present on a machine.

syslog Is a server which receives system logging messages. Currently, only the sendmail program uses this server.

telnetd Is a server for the DARPA standard TELNET protocol.

tftpd Is a server for the DARPA Trivial File Transfer Protocol.

trpt Is a program used in debugging TCP. Trpt transliterates protocol trace information recorded by TCP in a circular buffer in kernel memory.

tunefs Is a program for modifying certain parameters in the super block of file systems.

vipw Is no longer a shell script and properly interacts with passwd, chsh, and chfn in locking the password file.

An introduction to the C shell

William Joy

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

ABSTRACT

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

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

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

November 8, 1980

---

†UNIX is a Trademark of Bell Laboratories.

An introduction to the C shell

William Joy

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

Introduction

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

In addition to this document, you will want to refer to a copy of the UNIX programmer's manual. The *csh* documentation in the manual provides a full description of all features of the shell and is a final reference for questions about the shell.

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

Acknowledgements

Numerous people have provided good input about previous versions of *csh* and aided in its debugging and in the debugging of its documentation. I would especially like to thank Michael Ubell who made the crucial observation that history commands could be done well over the word structure of input text, and implemented a prototype history mechanism in an older version of the shell. Eric Allman has also provided a large number of useful comments on the shell, helping to unify those concepts which are present and to identify and eliminate useless and marginally useful features. Mike O'Brien suggested the pathname hashing mechanism which speeds command execution. Jim Kulp added the job control and directory stack primitives and added their documentation to this introduction.

1. Terminal usage of the shell

1.1. The basic notion of commands

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

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

```
mail bill
```

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

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

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

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

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

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

An example of a useful command you can execute now is the *tset* command, which sets the default *erase* and *kill* characters on your terminal — the erase character erases the last character you typed and the kill character erases the entire line you have entered so far. By default, the erase character is '#' and the kill character is '@'. Most people who use CRT displays prefer to use the backspace (↑H) character as their erase character since it is then easier to see what you have typed so far. You can make this be true by typing

tset −e

which tells the program *tset* to set the erase character, and its default setting for this character is a backspace.

1.2. Flag arguments

A useful notion in UNIX is that of a *flag* argument. While many arguments to commands specify file names or user names some arguments rather specify an optional capability of the command which you wish to invoke. By convention, such arguments begin with the character '−' (hyphen). Thus the command

ls

will produce a list of the files in the current *working directory*. The option −*s* is the size option, and

ls −s

causes *ls* to also give, for each file the size of the file in blocks of 512 characters. The manual section for each command in the UNIX reference manual gives the available options for each command. The *ls* command has a large number of useful and interesting options. Most other commands have either no options or only one or two options. It is hard to remember options of commands which are not used very frequently, so most UNIX utilities perform only one or two functions rather than having a large number of hard to remember options.

1.3. Output to files

Commands that normally read input or write output on the terminal can also be executed with this input and/or output done to a file.

Thus suppose we wish to save the current date in a file called 'now'. The command

date

will print the current date on our terminal. This is because our terminal is the default *standard output* for the date command and the date command prints the date on its standard output. The shell lets us *redirect* the *standard output* of a command through a notation using the *metacharacter* '>' and the name of the file where output is to be placed. Thus the command

date > now

runs the *date* command such that its standard output is the file 'now' rather than the terminal. Thus this command places the current date and time into the file 'now'. It is important to know that the *date* command was unaware that its output was going to a file rather than to the terminal. The shell performed this *redirection* before the command began executing.

One other thing to note here is that the file 'now' need not have existed before the *date* command was executed; the shell would have created the file if it did not exist. And if the file did exist? If it had existed previously these previous contents would have been discarded! A shell option *noclobber* exists to prevent this from happening accidentally; it is discussed in section 2.2.

The system normally keeps files which you create with '>' and all other files. Thus the default is for files to be permanent. If you wish to create a file which will be removed automatically, you can begin its name with a '#' character, this 'scratch' character denotes the fact that the file will be a scratch file.* The system will remove such files after a couple of days, or

*Note that if your erase character is a '#', you will have to precede the '#' with a '\'. The fact that the '#' character is the old (pre-CRT) standard erase character means that it seldom appears in a file name, and allows this convention to be used for scratch files. If you are using a CRT, your erase character should be a ↑H, as we demonstrated in section 1.1 how this could be set up.

sooner if file space becomes very tight. Thus, in running the *date* command above, we don't really want to save the output forever, so we would more likely do

    date > #now

1.4. Metacharacters in the shell

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

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

1.5. Input from files; pipelines

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

    sort < data

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

    sort data

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

We should note that if we just typed

    sort

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

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

    ls −s

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

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

    ls −s | sort −n

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

ls −s | sort −n −r | head −5

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

The notation introduced above is called the *pipe* mechanism. Commands separated by '|' characters are connected together by the shell and the standard output of each is run into the standard input of the next. The leftmost command in a pipeline will normally take its standard input from the terminal and the rightmost will place its standard output on the terminal. Other examples of pipelines will be given later when we discuss the history mechanism; one important use of pipes which is illustrated there is in the routing of information to the line printer.

1.6. Filenames

Many commands to be executed will need the names of files as arguments. UNIX *pathnames* consist of a number of *components* separated by '/'. Each component except the last names a directory in which the next component resides, in effect specifying the *path* of directories to follow to reach the file. Thus the pathname

/etc/motd

specifies a file in the directory 'etc' which is a subdirectory of the *root* directory '/'. Within this directory the file named is 'motd' which stands for 'message of the day'. A *pathname* that begins with a slash is said to be an *absolute* pathname since it is specified from the absolute top of the entire directory hierarchy of the system (the *root*). *Pathnames* which do not begin with '/' are interpreted as starting in the current *working directory*, which is, by default, your *home* directory and can be changed dynamically by the *cd* change directory command. Such pathnames are said to be *relative* to the working directory since they are found by starting in the working directory and descending to lower levels of directories for each *component* of the pathname. If the pathname contains no slashes at all then the file is contained in the working directory itself and the pathname is merely the name of the file in this directory. Absolute pathnames have no relation to the working directory.

Most filenames consist of a number of alphanumeric characters and '.'s (periods). In fact, all printing characters except '/' (slash) may appear in filenames. It is inconvenient to have most non-alphabetic characters in filenames because many of these have special meaning to the shell. The character '.' (period) is not a shell-metacharacter and is often used to separate the *extension* of a file name from the base of the name. Thus

prog.c prog.o prog.errs prog.output

are four related files. They share a *base* portion of a name (a base portion being that part of the name that is left when a trailing '.' and following characters which are not '.' are stripped off). The file 'prog.c' might be the source for a C program, the file 'prog.o' the corresponding object file, the file 'prog.errs' the errors resulting from a compilation of the program and the file 'prog.output' the output of a run of the program.

If we wished to refer to all four of these files in a command, we could use the notation

prog.*

This word is expanded by the shell, before the command to which it is an argument is executed, into a list of names which begin with 'prog.'. The character '*' here matches any sequence (including the empty sequence) of characters in a file name. The names which match are alphabetically sorted and placed in the *argument list* of the command. Thus the command

echo prog.*

will echo the names

prog.c prog.errs prog.o prog.output

Note that the names are in sorted order here, and a different order than we listed them above. The *echo* command receives four words as arguments, even though we only typed one word as as argument directly. The four words were generated by *filename expansion* of the one input word.

Other notations for *filename expansion* are also available. The character '?' matches any single character in a filename. Thus

echo ? ?? ???

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

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

prog.[co]

will match

prog.c prog.o

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

chap.[1−5]

might match files

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

if they existed. This is shorthand for

chap.[12345]

and otherwise equivalent.

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

No match.

and does not execute the command.

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

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

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

cp thatfile ~

the shell will expand this command to

cp thatfile /usr/bill

since my home directory is /usr/bill.

There also exists a mechanism using the characters '{' and '}' for abbreviating a set of words which have common parts but cannot be abbreviated by the above mechanisms because they are not files, are the names of files which do not yet exist, are not thus conveniently described. This mechanism will be described much later, in section 4.2, as it is used less frequently.

1.7. Quotation

We have already seen a number of metacharacters used by the shell. These metacharacters pose a problem in that we cannot use them directly as parts of words. Thus the command

echo *

will not echo the character '*'. It will either echo an sorted list of filenames in the current *working directory*, or print the message 'No match' if there are no files in the working directory.

The recommended mechanism for placing characters which are neither numbers, digits, '/', '.' or '−' in an argument word to a command is to enclose it with single quotation characters ''', i.e.

echo '*'

There is one special character '!' which is used by the *history* mechanism of the shell and which cannot be *escaped* by placing it within ''' characters. It and the character ''' itself can be preceded by a single '\' to prevent their special meaning. Thus

echo \'\!

prints

'!

These two mechanisms suffice to place any printing character into a word which is an argument to a shell command. They can be combined, as in

echo \''*'

which prints

'*

since the first '\' escaped the first ''' and the '*' was enclosed between ''' characters.

1.8. Terminating commands

When you are executing a command and the shell is waiting for it to complete there are several ways to force it to stop. For instance if you type the command

cat /etc/passwd

the system will print a copy of a list of all users of the system on your terminal. This is likely to continue for several minutes unless you stop it. You can send an INTERRUPT *signal* to the *cat* command by typing the DEL or RUBOUT key on your terminal.* Since *cat* does not take any precautions to avoid or otherwise handle this signal the INTERRUPT will cause it to terminate. The shell notices that *cat* has terminated and prompts you again with '% '. If you hit INTERRUPT

*Many users use *stty*(1) to change the interrupt character to ↑C.

again, the shell will just repeat its prompt since it handles INTERRUPT signals and chooses to continue to execute commands rather than terminating like *cat* did, which would have the effect of logging you out.

Another way in which many programs terminate is when they get an end-of-file from their standard input. Thus the *mail* program in the first example above was terminated when we typed a ↑D which generates an end-of-file from the standard input. The shell also terminates when it gets an end-of-file printing 'logout'; UNIX then logs you off the system. Since this means that typing too many ↑D's can accidentally log us off, the shell has a mechanism for preventing this. This *ignoreeof* option will be discussed in section 2.2.

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

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

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

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

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

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

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

In this example someone was sending a message to Harold and forgot the name of the file he wanted to mention. The mail command was suspended by typing ↑Z. When the shell noticed

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

If you write or run programs which are not fully debugged then it may be necessary to stop them somewhat ungracefully. This can be done by sending them a QUIT signal, sent by typing a ↑\. This will usually provoke the shell to produce a message like:

Quit (Core dumped)

indicating that a file 'core' has been created containing information about the program 'a.out's state when it terminated due to the QUIT signal. You can examine this file yourself, or forward information to the maintainer of the program telling him/her where the *core file* is.

If you run background commands (as explained in section 2.6) then these commands will ignore INTERRUPT and QUIT signals at the terminal. To stop them you must use the *kill* command. See section 2.6 for an example.

If you want to examine the output of a command without having it move off the screen as the output of the

cat /etc/passwd

command will, you can use the command

more /etc/passwd

The *more* program pauses after each complete screenful and types '−−More−−' at which point you can hit a space to get another screenful, a return to get another line, or a 'q' to end the *more* program. You can also use more as a filter, i.e.

cat /etc/passwd | more

works just like the more simple more command above.

For stopping output of commands not involving *more* you can use the ↑S key to stop the typeout. The typeout will resume when you hit ↑Q or any other key, but ↑Q is normally used because it only restarts the output and does not become input to the program which is running. This works well on low-speed terminals, but at 9600 baud it is hard to type ↑S and ↑Q fast enough to paginate the output nicely, and a program like *more* is usually used.

An additional possibility is to use the ↑O flush output character; when this character is typed, all output from the current command is thrown away (quickly) until the next input read occurs or until the next shell prompt. This can be used to allow a command to complete without having to suffer through the output on a slow terminal; ↑O is a toggle, so flushing can be turned off by typing ↑O again while output is being flushed.

1.9. What now?

We have so far seen a number of mechanisms of the shell and learned a lot about the way in which it operates. The remaining sections will go yet further into the internals of the shell, but you will surely want to try using the shell before you go any further. To try it you can log in to UNIX and type the following command to the system:

chsh myname /bin/csh

Here 'myname' should be replaced by the name you typed to the system prompt of 'login:' to get onto the system. Thus I would use 'chsh bill /bin/csh'. **You only have to do this once; it**

takes effect at next login. You are now ready to try using *csh*.

Before you do the 'chsh' command, the shell you are using when you log into the system is '/bin/sh'. In fact, much of the above discussion is applicable to '/bin/sh'. The next section will introduce many features particular to *csh* so you should change your shell to *csh* before you begin reading it.

2. Details on the shell for terminal users

2.1. Shell startup and termination

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

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

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

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

```
biff y
```

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

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

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

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

When the 'mail' and 'msgs' programs finish, the shell will finish processing my *.login* file and begin reading commands from the terminal, prompting for each with '% '. When I log off (by giving the *logout* command) the shell will print 'logout' and execute commands from the file '.logout' if it exists in my home directory. After that the shell will terminate and UNIX will log me off the system. If the system is not going down, I will receive a new login message. In

any case, after the 'logout' message the shell is committed to terminating and will take no further input from my terminal.

2.2. Shell variables

The shell maintains a set of *variables*. We saw above the variables *history* and *time* which had values '10' and '15'. In fact, each shell variable has as value an array of zero or more *strings*. Shell variables may be assigned values by the set command. It has several forms, the most useful of which was given above and is

```
set name=value
```

Shell variables may be used to store values which are to be used in commands later through a substitution mechanism. The shell variables most commonly referenced are, however, those which the shell itself refers to. By changing the values of these variables one can directly affect the behavior of the shell.

One of the most important variables is the variable *path*. This variable contains a sequence of directory names where the shell searches for commands. The *set* command with no arguments shows the value of all variables currently defined (we usually say *set)* in the shell. The default value for path will be shown by *set* to be

```
% set
argv      ()
cwd       /usr/bill
home      /usr/bill
path      (. /usr/ucb /bin /usr/bin)
prompt    %
shell     /bin/csh
status    0
term      c100rv4pna
user      bill
%
```

This output indicates that the variable path points to the current directory '.' and then '/usr/ucb', '/bin' and '/usr/bin'. Commands which you may write might be in '.' (usually one of your directories). Commands developed at Berkeley, live in '/usr/ucb' while commands developed at Bell Laboratories live in '/bin' and '/usr/bin'.

A number of locally developed programs on the system live in the directory '/usr/local'. If we wish that all shells which we invoke to have access to these new programs we can place the command

```
set path=(. /usr/ucb /bin /usr/bin /usr/local)
```

in our file *.cshrc* in our home directory. Try doing this and then logging out and back in and do

```
set
```

again to see that the value assigned to *path* has changed.

One thing you should be aware of is that the shell examines each directory which you insert into your path and determines which commands are contained there. Except for the current directory '.', which the shell treats specially, this means that if commands are added to a directory in your search path after you have started the shell, they will not necessarily be found by the shell. If you wish to use a command which has been added in this way, you should give the command

```
rehash
```

to the shell, which will cause it to recompute its internal table of command locations, so that it will find the newly added command. Since the shell has to look in the current directory '.' on

each command, placing it at the end of the path specification usually works equivalently and reduces overhead.

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

```
set ignoreeof
```

and to unset it do

```
unset ignoreeof
```

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

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

```
> filename
```

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

```
set noclobber
```

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

```
date > now
```

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

```
date >! now
```

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

2.3. The shell's history list

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

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

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

```
% cat bug.c
main()

{
        printf("hello);
}
% cc !$
cc bug.c
"bug.c", line 4: newline in string or char constant
"bug.c", line 5: syntax error
% ed !$
ed bug.c
29
4s/);/"&/p
        printf("hello");
w
30
q
% !c
cc bug.c
% a.out
hello% !e
ed bug.c
30
4s/lo/lo\\n/p
        printf("hello\n");
w
32
q
% !c -o bug
cc bug.c -o bug
% size a.out bug
a.out: 2784+364+1028 = 4176b = 0x1050b
bug: 2784+364+1028 = 4176b = 0x1050b
% ls -l !*
ls -l a.out bug
-rwxr-xr-x 1 bill       3932 Dec 19 09:41 a.out
-rwxr-xr-x 1 bill       3932 Dec 19 09:42 bug
% bug
hello
% num bug.c | spp
spp: Command not found.
% ↑spp↑ssp
num bug.c | ssp
   1  main()
   3  {
   4          printf("hello\n");
   5  }
% !! | lpr
num bug.c | ssp | lpr
%
```

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

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

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

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

2.4. Aliases

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

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

```
alias mail newmail
```

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

```
mail bill
```

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

```
alias ls ls −s
```

or even

```
alias dir ls −s
```

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

```
dir ~bill
```

then the shell will translate this to

```
ls −s /mnt/bill
```

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

arguments to the original command are to be substituted using the facilities of the history mechanism. Thus the definition

    alias cd 'cd \!* ; ls '

would do an *ls* command after each change directory *cd* command. We enclosed the entire alias definition in '" characters to prevent most substitutions from occurring and the character ';' from being recognized as a metacharacter. The '!' here is escaped with a '\' to prevent it from being interpreted when the alias command is typed in. The '\!*' here substitutes the entire argument list to the pre-aliasing *cd* command, without giving an error if there were no arguments. The ';' separating commands is used here to indicate that one command is to be done and then the next. Similarly the definition

    alias whois 'grep \!↑ /etc/passwd'

defines a command which looks up its first argument in the password file.

Warning: The shell currently reads the *.cshrc* file each time it starts up. If you place a large number of commands there, shells will tend to start slowly. A mechanism for saving the shell environment after reading the *.cshrc* file and quickly restoring it is under development, but for now you should try to limit the number of aliases you have to a reasonable number... 10 or 15 is reasonable, 50 or 60 will cause a noticeable delay in starting up shells, and make the system seem sluggish when you execute commands from within the editor and other programs.

2.5. More redirection; >> and >&

There are a few more notations useful to the terminal user which have not been introduced yet.

In addition to the standard output, commands also have a *diagnostic output* which is normally directed to the terminal even when the standard output is redirected to a file or a pipe. It is occasionally desirable to direct the diagnostic output along with the standard output. For instance if you want to redirect the output of a long running command into a file and wish to have a record of any error diagnostic it produces you can do

    command >& file

The '>&' here tells the shell to route both the diagnostic output and the standard output into 'file'. Similarly you can give the command

    command |& lpr

to route both standard and diagnostic output through the pipe to the line printer daemon *lpr*.#

Finally, it is possible to use the form

    command >> file

to place output at the end of an existing file.†

#A command form

    command >&! file

exists, and is used when *noclobber* is set and *file* already exists.

†If *noclobber* is set, then an error will result if *file* does not exist, otherwise the shell will create *file* if it doesn't exist. A form

    command >>! file

makes it not be an error for file to not exist when *noclobber* is set.

2.6. Jobs; Background, Foreground, or Suspended

When one or more commands are typed together as a pipeline or as a sequence of commands separated by semicolons, a single *job* is created by the shell consisting of these commands together as a unit. Single commands without pipes or semicolons create the simplest jobs. Usually, every line typed to the shell creates a job. Some lines that create jobs (one per line) are

```
sort < data
ls −s | sort −n | head −5
mail harold
```

If the metacharacter '&' is typed at the end of the commands, then the job is started as a *background* job. This means that the shell does not wait for it to complete but immediately prompts and is ready for another command. The job runs *in the background* at the same time that normal jobs, called *foreground* jobs, continue to be read and executed by the shell one at a time. Thus

```
du > usage &
```

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

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

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

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

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

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

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

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

```
% du > usage
↑Z
Stopped
%
```

'Stopped' message is typed by the shell when it notices that the *du* program stopped. For background jobs, using the *stop* command, it is

```
% sort usage &
[1] 2345
% stop %1
[1] + Stopped (signal)        sort usage
%
```

Suspending foreground jobs can be very useful when you need to temporarily change what you are doing (execute other commands) and then return to the suspended job. Also, foreground jobs can be suspended and then continued as background jobs using the *bg* command, allowing you to continue other work and stop waiting for the foreground job to finish. Thus

```
% du > usage
↑Z
Stopped
% bg
[1] du > usage &
%
```

starts 'du' in the foreground, stops it before it finishes, then continues it in the background allowing more foreground commands to be executed. This is especially helpful when a foreground job ends up taking longer than you expected and you wish you had started it in the backgound in the beginning.

All *job control* commands can take an argument that identifies a particular job. All job name arguments begin with the character '%', since some of the job control commands also accept process numbers (printed by the *ps* command.) The default job (when no argument is given) is called the *current* job and is identified by a '+' in the output of the *jobs* command, which shows you which jobs you have. When only one job is stopped or running in the background (the usual case) it is always the current job thus no argument is needed. If a job is stopped while running in the foreground it becomes the *current* job and the existing current job becomes the *previous* job — identified by a '−' in the output of *jobs*. When the current job terminates, the previous job becomes the current job. When given, the argument is either '%−' (indicating the previous job); '%#', where # is the job number; '%pref' where pref is some

unique prefix of the command name and arguments of one of the jobs; or '%?' followed by some string found in only one of the jobs.

The *jobs* command types the table of jobs, giving the job number, commands and status ('Stopped' or 'Running') of each backgound or suspended job. With the '−l' option the process numbers are also typed.

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

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

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

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

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

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

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

```
ed bigfile
w
120000
q
%
```

So after the 's' command was issued, the 'ed' job was stopped with ↑Z and then put in the background using *bg*. Some time later when the 's' command was finished, *ed* tried to read another command and was stopped because jobs in the backgound cannot read from the terminal. The *fg* command returned the 'ed' job to the foreground where it could once again accept commands from the terminal.

The command

```
stty tostop
```

causes all background jobs run on your terminal to stop when they are about to write output to the terminal. This prevents messages from background jobs from interrupting foreground job output and allows you to run a job in the background without losing terminal output. It also can be used for interactive programs that sometimes have long periods without interaction. Thus each time it outputs a prompt for more input it will stop before the prompt. It can then be run in the foreground using *fg*, more input can be given and, if necessary stopped and returned to the background. This *stty* command might be a good thing to put in your *.login* file if you do not like output from background jobs interrupting your work. It also can reduce the need for redirecting the output of background jobs if the output is not very big:

```
% stty tostop
% wc hugefile &
[1] 10387
% ed text
. . . some time later
q
[1] Stopped (tty output)      wc hugefile
% fg wc
wc hugefile
  13371  30123  302577
% stty −tostop
```

Thus after some time the 'wc' command, which counts the lines, words and characters in a file, had one line of output. When it tried to write this to the terminal it stopped. By restarting it in the foreground we allowed it to write on the terminal exactly when we were ready to look at its output. Programs which attempt to change the mode of the terminal will also block, whether or not *tostop* is set, when they are not in the foreground, as it would be very unpleasant to have a background job change the state of the terminal.

Since the *jobs* command only prints jobs started in the currently executing shell, it knows nothing about background jobs started in other login sessions or within shell files. The *ps* can be used in this case to find out about background jobs not started in the current shell.

2.7. Working Directories

As mentioned in section 1.6, the shell is always in a particular *working directory*. The 'change directory' command *chdir* (its short form *cd* may also be used) changes the working directory of the shell, that is, changes the directory you are located in.

It is useful to make a directory for each project you wish to work on and to place all files related to that project in that directory. The 'make directory' command, *mkdir*, creates a new directory. The *pwd* ('print working directory') command reports the absolute pathname of the working directory of the shell, that is, the directory you are located in. Thus in the example below:

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

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

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

```
cd
```

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

```
cd ..
```

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

```
cd ../programs
```

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

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

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

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

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

Since the shell remembers the working directory in which each job was started, it warns you when you might be confused by restarting a job in the foreground which has a different working directory than the current working directory of the shell. Thus if you start a background job, then change the shell's working directory and then cause the background job to run in the foreground, the shell warns you that the working directory of the currently running foreground job is different from that of the shell.

```
% dirs −l
/mnt/bill
% cd myproject
% dirs
~/myproject
% ed prog.c
1143
↑Z
Stopped
% cd ..
% ls
myproject
textfile
% fg
ed prog.c (wd: ~/myproject)
```

This way the shell warns you when there is an implied change of working directory, even though no cd command was issued. In the above example the 'ed' job was still in '/mnt/bill/project' even though the shell had changed to '/mnt/bill'. A similar warning is given when such a foreground job terminates or is suspended (using the STOP signal) since the return to the shell again implies a change of working directory.

```
% fg
ed prog.c (wd: ~/myproject)
 . . . after some editing
q
(wd now: ~)
%
```

These messages are sometimes confusing if you use programs that change their own working directories, since the shell only remembers which directory a job is started in, and assumes it stays there. The '−l' option of *jobs* will type the working directory of suspended or background jobs when it is different from the current working directory of the shell.

2.8. Useful built-in commands

We now give a few of the useful built-in commands of the shell describing how they are used.

The *alias* command described above is used to assign new aliases and to show the existing aliases. With no arguments it prints the current aliases. It may also be given only one argument such as

```
alias ls
```

to show the current alias for, e.g., 'ls'.

The *echo* command prints its arguments. It is often used in *shell scripts* or as an interactive command to see what filename expansions will produce.

The *history* command will show the contents of the history list. The numbers given with the history events can be used to reference previous events which are difficult to reference using the contextual mechanisms introduced above. There is also a shell variable called *prompt*.

By placing a '!' character in its value the shell will there substitute the number of the current command in the history list. You can use this number to refer to this command in a history substitution. Thus you could

    set prompt='\! % '

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

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

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

Limits can be set, e.g.:

    limit coredumpsize 128k

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

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

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

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

    repeat 5 cat one >> five

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

    setenv TERM adm3a

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

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

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

    source .cshrc

can be used after editing in a change to the *.cshrc* file which you wish to take effect before the next time you login.

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

```
% time cp /etc/rc /usr/bill/rc
0.0u 0.1s 0:01 8% 2+1k 3+2io 1pf+0w
% time wc /etc/rc /usr/bill/rc
    52   178   1347 /etc/rc
    52   178   1347 /usr/bill/rc
   104   356   2694 total
0.1u 0.1s 0:00 13% 3+3k 5+3io 7pf+0w
%
```

indicates that the *cp* command used a negligible amount of user time (u) and about 1/10th of a system time (s); the elapsed time was 1 second (0:01), there was an average memory usage of 2k bytes of program space and 1k bytes of data space over the cpu time involved (2+1k); the program did three disk reads and two disk writes (3+2io), and took one page fault and was not swapped (1pf+0w). The word count command *wc* on the other hand used 0.1 seconds of user time and 0.1 seconds of system time in less than a second of elapsed time. The percentage '13%' indicates that over the period when it was active the command 'wc' used an average of 13 percent of the available CPU cycles of the machine.

The *unalias* and *unset* commands can be used to remove aliases and variable definitions from the shell, and *unsetenv* removes variables from the environment.

2.9. What else?

This concludes the basic discussion of the shell for terminal users. There are more features of the shell to be discussed here, and all features of the shell are discussed in its manual pages. One useful feature which is discussed later is the *foreach* built-in command which can be used to run the same command sequence with a number of different arguments.

If you intend to use UNIX a lot you you should look through the rest of this document and the shell manual pages to become familiar with the other facilities which are available to you.

3. Shell control structures and command scripts

3.1. Introduction

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

3.2. Make

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

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

3.3. Invocation and the argv variable

A *csh* command script may be interpreted by saying

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

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

If you make the file 'script' executable by doing

```
chmod 755 script
```

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

```
script
```

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

3.4. Variable substitution

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

```
echo $argv
```

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

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

```
$?name
```

expands to '1' if name is *set* or to '0' if name is not *set.* It is the fundamental mechanism used

for checking whether particular variables have been assigned values. All other forms of reference to undefined variables cause errors.

The notation

```
$#name
```

expands to the number of elements in the variable *name*. Thus

```
% set argv=(a b c)
% echo $?argv
1
% echo $#argv
3
% unset argv
% echo $?argv
0
% echo $argv
Undefined variable: argv.
%
```

It is also possible to access the components of a variable which has several values. Thus

```
$argv[1]
```

gives the first component of *argv* or in the example above 'a'. Similarly

```
$argv[$#argv]
```

would give 'c', and

```
$argv[1-2]
```

would give 'a b'. Other notations useful in shell scripts are

```
$n
```

where *n* is an integer as a shorthand for

```
$argv[n]
```

the *n th* parameter and

```
$*
```

which is a shorthand for

```
$argv
```

The form

```
$$
```

expands to the process number of the current shell. Since this process number is unique in the system it can be used in generation of unique temporary file names. The form

```
$<
```

is quite special and is replaced by the next line of input read from the shell's standard input (not the script it is reading). This is useful for writing shell scripts that are interactive, reading commands from the terminal, or even writing a shell script that acts as a filter, reading lines from its input file. Thus the sequence

```
echo 'yes or no?\c'
set a=($<)
```

would write out the prompt 'yes or no?' without a newline and then read the answer into the

variable 'a'. In this case '$#a' would be '0' if either a blank line or end-of-file (↑D) was typed.

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

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

3.5. Expressions

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

The shell also allows file enquiries of the form

−? filename

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

−e filename

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

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

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

3.6. Sample shell script

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

```
% cat copyc
#
# Copyc copies those C programs in the specified list
# to the directory ~/backup if they differ from the files
# already in ~/backup
#
set noglob
foreach i ($argv)

        if ($i !~ *.c) continue  # not a .c file so do nothing

        if (! −r ~/backup/$i:t) then
                echo $i:t not in backup... not cp\'ed
                continue
        endif

        cmp −s $i ~/backup/$i:t # to set $status

        if ($status != 0) then
                echo new backup of $i
                cp $i ~/backup/$i:t
        endif
end
```

This script makes use of the *foreach* command, which causes the shell to execute the commands between the *foreach* and the matching *end* for each of the values given between '(' and ')' with the named variable, in this case 'i' set to successive values in the list. Within this loop we may use the command *break* to stop executing the loop and *continue* to prematurely terminate one iteration and begin the next. After the *foreach* loop the iteration variable (*i* in this case) has the value at the last iteration.

We set the variable *noglob* here to prevent filename expansion of the members of *argv*. This is a good idea, in general, if the arguments to a shell script are filenames which have already been expanded or if the arguments may contain filename expansion metacharacters. It is also possible to quote each use of a '$' variable expansion, but this is harder and less reliable.

The other control construct used here is a statement of the form

```
if ( expression ) then
        command
        ...
endif
```

The placement of the keywords here is **not** flexible due to the current implementation of the shell.†

---

†The following two formats are not currently acceptable to the shell:

```
if ( expression )               # Won't work!
then
        command
        ...
endif
```

and

```
if ( expression ) then command endif            # Won't work
```

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

```
if ( expression ) command
```

which can be written

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

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

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

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

else
        commands
endif
```

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

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

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

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

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

does not do what one would expect.

3.7. Other control structures

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

```
while ( expression )
        commands
end
```

and

```
switch ( word )

case str1:
        commands
        breaksw

...

case strn:
        commands
        breaksw

default:
        commands
        breaksw

endsw
```

For details see the manual section for *csh*. C programmers should note that we use *breaksw* to exit from a *switch* while *break* exits a *while* or *foreach* loop. A common mistake to make in *csh* scripts is to use *break* rather than *breaksw* in switches.

Finally, *csh* allows a *goto* statement, with labels looking like they do in C, i.e.:

```
loop:
        commands
        goto loop
```

3.8. Supplying input to commands

Commands run from shell scripts receive by default the standard input of the shell which is running the script. This is different from previous shells running under UNIX. It allows shell scripts to fully participate in pipelines, but mandates extra notation for commands which are to take inline data.

Thus we need a metanotation for supplying inline data to commands in shell scripts. As an example, consider this script which runs the editor to delete leading blanks from the lines in each argument file

```
% cat deblank
# deblank -- remove leading blanks
foreach i ($argv)
ed - $i << 'EOF'
1,$s/↑[ ]*//
w
q
'EOF'
end
%
```

The notation '<< 'EOF'' means that the standard input for the *ed* command is to come from the text in the shell script file up to the next line consisting of exactly ''EOF''. The fact that the 'EOF' is enclosed in ''' characters, i.e. quoted, causes the shell to not perform variable substitution on the intervening lines. In general, if any part of the word following the '<<' which the shell uses to terminate the text to be given to the command is quoted then these substitutions will not be performed. In this case since we used the form '1,$' in our editor script we needed to insure that this '$' was not variable substituted. We could also have insured this by preceding the '$' here with a '\', i.e.:

```
1,\$s/↑[ ]*//
```

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

3.9. Catching interrupts

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

```
onintr label
```

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

```
exit(1)
```

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

3.10. What else?

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

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

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

4. Other, less commonly used, shell features

4.1. Loops at the terminal; variables as vectors

It is occasionally useful to use the *foreach* control structure at the terminal to aid in performing a number of similar commands. For instance, there were at one point three shells in use on the Cory UNIX system at Cory Hall, '/bin/sh', '/bin/nsh', and '/bin/csh'. To count the number of persons using each shell one could have issued the commands

```
% grep −c csh$ /etc/passwd
27
% grep −c nsh$ /etc/passwd
128
% grep −c −v sh$ /etc/passwd
430
%
```

Since these commands are very similar we can use *foreach* to do this more easily.

```
% foreach i ('sh$' 'csh$' '−v sh$')
? grep −c $i /etc/passwd
? end
27
128
430
%
```

Note here that the shell prompts for input with '? ' when reading the body of the loop.

Very useful with loops are variables which contain lists of filenames or other words. You can, for example, do

```
% set a=(`ls`)
% echo $a
csh.n csh.rm
% ls
csh.n
csh.rm
% echo $#a
2
%
```

The *set* command here gave the variable *a* a list of all the filenames in the current directory as value. We can then iterate over these names to perform any chosen function.

The output of a command within '`' characters is converted by the shell to a list of words. You can also place the '`' quoted string within '"' characters to take each (non-empty) line as a component of the variable; preventing the lines from being split into words at blanks and tabs. A modifier ':x' exists which can be used later to expand each component of the variable into another variable splitting it into separate words at embedded blanks and tabs.

4.2. Braces { ... } in argument expansion

Another form of filename expansion, alluded to before involves the characters '{' and '}'. These characters specify that the contained strings, separated by ',' are to be consecutively substituted into the containing characters and the results expanded left to right. Thus

```
A{str1,str2,...strn}B
```

expands to

Astr1B Astr2B ... AstrnB

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

A typical use of this would be

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

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

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

4.3. Command substitution

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

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

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

```
ex `grep −l TRACE *.c`
```

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

4.4. Other details not covered here

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

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

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

Appendix — Special characters

The following table lists the special characters of *csh* and the UNIX system, giving for each the section(s) in which it is discussed. A number of these characters also have special meaning in expressions. See the *csh* manual section for a complete list.

Syntactic metacharacters

;	2.4	separates commands to be executed sequentially
|	1.5	separates commands in a pipeline
( )	2.2,3.6	brackets expressions and variable values
&	2.5	follows commands to be executed without waiting for completion

Filename metacharacters

/	1.6	separates components of a file's pathname
?	1.6	expansion character matching any single character
*	1.6	expansion character matching any sequence of characters
[ ]	1.6	expansion sequence matching any single character from a set
~	1.6	used at the beginning of a filename to indicate home directories
{ }	4.2	used to specify groups of arguments with common parts

Quotation metacharacters

\	1.7	prevents meta-meaning of following single character
'	1.7	prevents meta-meaning of a group of characters
"	4.3	like ', but allows variable and command expansion

Input/output metacharacters

<	1.5	indicates redirected input
>	1.3	indicates redirected output

Expansion/substitution metacharacters

$	3.4	indicates variable substitution
!	2.3	indicates history substitution
:	3.6	precedes substitution modifiers
↑	2.3	used in special forms of history substitution
`	4.3	indicates command substitution

Other metacharacters

#	1.3,3.6	begins scratch file names; indicates shell comments
–	1.2	prefixes option (flag) arguments to commands
%	2.6	prefixes job name specifications

Glossary

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

```
man 1 pr
```

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

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

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

```
chdir paper
```

you can return to the parent directory by doing

```
chdir ..
```

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

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

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

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

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

```
echo a b c
```

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

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

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

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

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

bin — A directory containing binaries of programs and shell scripts to be executed is typically called a *bin* directory. The standard system *bin* directories are '/bin' containing the most heavily used commands and '/usr/bin' which contains most other user programs. Programs developed at UC Berkeley live in '/usr/ucb', while locally written programs live in '/usr/local'. Games are kept in the directory '/usr/games'. You can place binaries in any directory. If you wish to execute them often, the name of the directories should be a *component* of the variable *path*.

break — *Break* is a builtin command used to exit from loops within the control structure of the shell (3.7).

breaksw — The *breaksw* builtin command is used to exit from a *switch* control structure, like a *break* exits from loops (3.7).

builtin — A command executed directly by the shell is called a *builtin* command. Most commands in UNIX are not built into the shell, but rather exist as files in *bin* directories. These commands are accessible because the directories in which they reside are named in the *path* variable.

case — A *case* command is used as a label in a *switch* statement in the shell's control structure, similar to that of the language C. Details are given in the shell documentation 'csh(1)' (3.7).

cat — The *cat* program catenates a list of specified files on the *standard output*. It is usually used to look at the contents of a single file on the terminal, to 'cat a file' (1.8, 2.3).

cd — The *cd* command is used to change the *working directory*. With no arguments, *cd* changes your *working directory* to be your *home* directory (2.4, 2.7).

chdir — The *chdir* command is a synonym for *cd*. *Cd* is usually used because it is easier to type.

chsh — The *chsh* command is used to change the shell which you use on UNIX. By default, you use an different version of the shell which resides in '/bin/sh'. You can change your shell to '/bin/csh' by doing

```
chsh your-login-name /bin/csh
```

Thus I would do

```
chsh bill /bin/csh
```

It is only necessary to do this once. The next time you log in to UNIX after doing this command, you will be using *csh* rather than the shell in '/bin/sh' (1.9).

cmp — *Cmp* is a program which compares files. It is usually used on binary files, or to see if two files are identical (3.6). For comparing text files the program *diff*, described in 'diff (1)' is used.

command — A function performed by the system, either by the shell (a builtin *command*) or by a program residing in a file in a directory within the UNIX system, is called a *command* (1.1).

command name — When a command is issued, it consists of a *command name*, which is the first word of the command, followed by arguments. The convention on UNIX is that the first word of a command names the function to be performed (1.1).

command substitution
: The replacement of a command enclosed in '`' characters by the text output by that command is called *command substitution* (4.3).

component
: A part of a *pathname* between '/' characters is called a *component* of that *pathname*. A variable which has multiple strings as value is said to have several *components*; each string is a *component* of the variable.

continue
: A builtin command which causes execution of the enclosing *foreach* or *while* loop to cycle prematurely. Similar to the *continue* command in the programming language C (3.6).

control-
: Certain special characters, called *control* characters, are produced by holding down the CONTROL key on your terminal and simultaneously pressing another character, much like the SHIFT key is used to produce upper case characters. Thus *control-c* is produced by holding down the CONTROL key while pressing the 'c' key. Usually UNIX prints an up-arrow (↑) followed by the corresponding letter when you type a *control* character (e.g. '↑C' for *control-c* (1.8).

core dump
: When a program terminates abnormally, the system places an image of its current state in a file named 'core'. This *core dump* can be examined with the system debugger 'adb(1)' or 'sdb(1)' in order to determine what went wrong with the program (1.8). If the shell produces a message of the form

    Illegal instruction (core dumped)

  (where 'Illegal instruction' is only one of several possible messages), you should report this to the author of the program or a system administrator, saving the 'core' file.

cp
: The *cp* (copy) program is used to copy the contents of one file into another file. It is one of the most commonly used UNIX commands (1.6).

csh
: The name of the shell program that this document describes.

.cshrc
: The file *.cshrc* in your *home* directory is read by each shell as it begins execution. It is usually used to change the setting of the variable *path* and to set *alias* parameters which are to take effect globally (2.1).

cwd
: The *cwd* variable in the shell holds the *absolute pathname* of the current *working directory*. It is changed by the shell whenever your current *working directory* changes and should not be changed otherwise (2.2).

date
: The *date* command prints the current date and time (1.3).

debugging
: *Debugging* is the process of correcting mistakes in programs and shell scripts. The shell has several options and variables which may be used to aid in shell *debugging* (4.4).

default:
: The label *default:* is used within shell *switch* statements, as it is in the C language to label the code to be executed if none of the *case* labels matches the value switched on (3.7).

DELETE
: The DELETE or RUBOUT key on the terminal normally causes an interrupt to be sent to the current job. Many users change the interrupt character to be ↑C.

detached
: A command that continues running in the *background* after you logout is said to be *detached*.

diagnostic
: An error message produced by a program is often referred to as a *diagnostic*. Most error messages are not written to the *standard output*, since that is often directed away from the terminal (1.3, 1.5). Error messsages are instead written to the *diagnostic output* which may be directed away from the terminal, but usually is not. Thus *diagnostics* will usually appear on the terminal (2.5).

directory — A structure which contains files. At any time you are in one particular *directory* whose names can be printed by the command *pwd*. The *chdir* command will change you to another *directory*, and make the files in that *directory* visible. The *directory* in which you are when you first login is your *home* directory (1.1, 2.7).

directory stack — The shell saves the names of previous *working directories* in the *directory stack* when you change your current *working directory* via the *pushd* command. The *directory stack* can be printed by using the *dirs* command, which includes your current *working directory* as the first directory name on the left (2.7).

dirs — The *dirs* command prints the shell's *directory stack* (2.7).

du — The *du* command is a program (described in 'du(1)') which prints the number of disk blocks is all directories below and including your current *working directory* (2.6).

echo — The *echo* command prints its arguments (1.6, 3.6).

else — The *else* command is part of the 'if-then-else-endif' control command construct (3.6).

endif — If an *if* statement is ended with the word *then*, all lines following the *if* up to a line starting with the word *endif* or *else* are executed if the condition between parentheses after the *if* is true (3.6).

EOF — An *end-of-file* is generated by the terminal by a control-d, and whenever a command reads to the end of a file which it has been given as input. Commands receiving input from a *pipe* receive an *end-of-file* when the command sending them input completes. Most commands terminate when they receive an *end-of-file*. The shell has an option to ignore *end-of-file* from a terminal input which may help you keep from logging out accidentally by typing too many control-d's (1.1, 1.8, 3.8).

escape — A character '\' used to prevent the special meaning of a metacharacter is said to *escape* the character from its special meaning. Thus

```
echo \*
```

will echo the character '*' while just

```
echo *
```

will echo the names of the file in the current directory. In this example, \ *escapes* '*' (1.7). There is also a non-printing character called *escape*, usually labelled ESC or ALTMODE on terminal keyboards. Some older UNIX systems use this character to indicate that output is to be *suspended*. Most systems use control-s to stop the output and control-q to start it.

/etc/passwd — This file contains information about the accounts currently on the system. It consists of a line for each account with fields separated by ':' characters (1.8). You can look at this file by saying

```
cat /etc/passwd
```

The commands *finger* and *grep* are often used to search for information in this file. See 'finger(1)', 'passwd(5)', and 'grep(1)' for more details.

exit — The *exit* command is used to force termination of a shell script, and is built into the shell (3.9).

exit status — A command which discovers a problem may reflect this back to the command (such as a shell) which invoked (executed) it. It does this by returning a non-zero number as its *exit status*, a status of zero being considered 'normal termination'. The *exit* command can be used to force a shell command script

to give a non-zero *exit status* (3.6).

expansion — The replacement of strings in the shell input which contain metacharacters by other strings is referred to as the process of *expansion.* Thus the replacement of the word '*' by a sorted list of files in the current directory is a 'filename expansion'. Similarly the replacement of the characters '!!' by the text of the last command is a 'history expansion'. *Expansions* are also referred to as *substitutions* (1.6, 3.4, 4.2).

expressions — *Expressions* are used in the shell to control the conditional structures used in the writing of shell scripts and in calculating values for these scripts. The operators available in shell *expressions* are those of the language C (3.5).

extension — Filenames often consist of a *base* name and an *extension* separated by the character '.'. By convention, groups of related files often share the same *root* name. Thus if 'prog.c' were a C program, then the object file for this program would be stored in 'prog.o'. Similarly a paper written with the '−me' nroff macro package might be stored in 'paper.me' while a formatted version of this paper might be kept in 'paper.out' and a list of spelling errors in 'paper.errs' (1.6).

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

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

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

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

```
ls −s
```

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

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

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

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

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

will print each line in the file */etc/passwd* which contains the string 'bill'.

Actually, *grep* scans for *regular expressions* in the sense of the editors 'ed(1)' and 'ex(1)'. *Grep* stands for 'globally find *regular expression* and print' (2.4).

head — The *head* command prints the first few lines of one or more files. If you have a bunch of files containing text which you are wondering about it is sometimes useful to run *head* with these files as arguments. This will usually show enough of what is in these files to let you decide which you are interested in (1.5).
Head is also used to describe the part of a *pathname* before and including the last '/' character. The *tail* of a *pathname* is the part after the last '/'. The ':h' and ':t' modifiers allow the *head* or *tail* of a *pathname* stored in a shell variable to be used (3.6).

history — The *history* mechanism of the shell allows previous commands to be repeated, possibly after modification to correct typing mistakes or to change the meaning of the command. The shell has a *history list* where these commands are kept, and a *history* variable which controls how large this list is (2.3).

home directory — Each user has a *home directory*, which is given in your entry in the password file, */etc/passwd*. This is the directory which you are placed in when you first login. The *cd* or *chdir* command with no arguments takes you back to this directory, whose name is recorded in the shell variable *home*. You can also access the *home directories* of other users in forming filenames using a *filename expansion* notation and the character '~' (1.6).

if — A conditional command within the shell, the *if* command is used in shell command scripts to make decisions about what course of action to take next (3.6).

ignoreeof — Normally, your shell will exit, printing 'logout' if you type a control-d at a prompt of '% '. This is the way you usually log off the system. You can *set* the *ignoreeof* variable if you wish in your *.login* file and then use the command *logout* to logout. This is useful if you sometimes accidentally type too many control-d characters, logging yourself off (2.2).

input — Many commands on UNIX take information from the terminal or from files which they then act on. This information is called *input*. Commands normally read for *input* from their *standard input* which is, by default, the terminal. This *standard input* can be redirected from a file using a shell metanotation with the character '<'. Many commands will also read from a file specified as argument. Commands placed in *pipelines* will read from the output of the previous command in the *pipeline*. The leftmost command in a *pipeline* reads from the terminal if you neither redirect its *input* nor give it a filename to use as *standard input*. Special mechanisms exist for supplying input to commands in shell scripts (1.5, 3.8).

interrupt — An *interrupt* is a signal to a program that is generated by hitting the RUBOUT or DELETE key (although users can and often do change the interrupt character, usually to ↑C). It causes most programs to stop execution. Certain programs, such as the shell and the editors, handle an *interrupt* in special ways, usually by stopping what they are doing and prompting for another command. While the shell is executing another command and waiting for it to finish, the shell does not listen to *interrupts*. The shell often wakes up when you hit *interrupt* because many commands die when they receive an *interrupt* (1.8, 3.9).

job — One or more commands typed on the same input line separated by '|' or ';' characters are run together and are called a *job*. Simple commands run by themselves without any '|' or ';' characters are the simplest *jobs*. *Jobs* are classified as *foreground*, *background*, or *suspended* (2.6).

job control — The builtin functions that control the execution of jobs are called *job control* commands. These are *bg, fg, stop, kill* (2.6).

job number — When each job is started it is assigned a small number called a *job number* which is printed next to the job in the output of the *jobs* command. This number, preceded by a '%' character, can be used as an argument to *job control* commands to indicate a specific job (2.6).

jobs — The *jobs* command prints a table showing jobs that are either running in the *background* or are *suspended* (2.6).

kill — A command which sends a signal to a job causing it to terminate (2.6).

.login — The file *.login* in your *home* directory is read by the shell each time you login to UNIX and the commands there are executed. There are a number of commands which are usefully placed here, especially *set* commands to the shell itself (2.1).

login shell — The shell that is started on your terminal when you login is called your *login shell.* It is different from other shells which you may run (e.g. on shell scripts) in that it reads the *.login* file before reading commands from the terminal and it reads the *.logout* file after you logout (2.1).

logout — The *logout* command causes a login shell to exit. Normally, a login shell will exit when you hit control-d generating an *end-of-file,* but if you have set *ignoreeof* in you *.login* file then this will not work and you must use *logout* to log off the UNIX system (2.8).

.logout — When you log off of UNIX the shell will execute commands from the file *.logout* in your *home* directory after it prints 'logout'.

lpr — The command *lpr* is the line printer daemon. The standard input of *lpr* spooled and printed on the UNIX line printer. You can also give *lpr* a list of filenames as arguments to be printed. It is most common to use *lpr* as the last component of a *pipeline* (2.3).

ls — The *ls* (list files) command is one of the most commonly used UNIX commands. With no argument filenames it prints the names of the files in the current directory. It has a number of useful *flag* arguments, and can also be given the names of directories as arguments, in which case it lists the names of the files in these directories (1.2).

mail — The *mail* program is used to send and receive messages from other UNIX users (1.1, 2.1).

make — The *make* command is used to maintain one or more related files and to organize functions to be performed on these files. In many ways *make* is easier to use, and more helpful than shell command scripts (3.2).

makefile — The file containing commands for *make* is called *makefile* (3.2).

manual — The *manual* often referred to is the 'UNIX programmer's manual'. It contains a number of sections and a description of each UNIX program. An online version of the *manual* is accessible through the *man* command. Its documentation can be obtained online via

```
man man
```

metacharacter — Many characters which are neither letters nor digits have special meaning either to the shell or to UNIX. These characters are called *metacharacters.* If it is necessary to place these characters in arguments to commands without them having their special meaning then they must be *quoted.* An example of a *metacharacter* is the character '>' which is used to indicate placement of output

into a file. For the purposes of the *history* mechanism, most unquoted *metacharacters* form separate words (1.4). The appendix to this user's manual lists the *metacharacters* in groups by their function.

mkdir — The *mkdir* command is used to create a new directory.

modifier — Substitutions with the *history* mechanism, keyed by the character '!' or of variables using the metacharacter '$', are often subjected to modifications, indicated by placing the character ':' after the substitution and following this with the *modifier* itself. The *command substitution* mechanism can also be used to perform modification in a similar way, but this notation is less clear (3.6).

more — The program *more* writes a file on your terminal allowing you to control how much text is displayed at a time. *More* can move through the file screenful by screenful, line by line, search forward for a string, or start again at the beginning of the file. It is generally the easiest way of viewing a file (1.8).

noclobber — The shell has a variable *noclobber* which may be set in the file *.login* to prevent accidental destruction of files by the '>' output redirection metasyntax of the shell (2.2, 2.5).

noglob — The shell variable *noglob* is set to suppress the *filename expansion* of arguments containing the metacharacters '~', '*', '?', '[' and ']' (3.6).

notify — The *notify* command tells the shell to report on the termination of a specific *background job* at the exact time it occurs as opposed to waiting until just before the next prompt to report the termination. The *notify* variable, if set, causes the shell to always report the termination of *background* jobs exactly when they occur (2.6).

onintr — The *onintr* command is built into the shell and is used to control the action of a shell command script when an *interrupt* signal is received (3.9).

output — Many commands in UNIX result in some lines of text which are called their *output*. This *output* is usually placed on what is known as the *standard output* which is normally connected to the user's terminal. The shell has a syntax using the metacharacter '>' for redirecting the *standard output* of a command to a file (1.3). Using the *pipe* mechanism and the metacharacter '|' it is also possible for the *standard output* of one command to become the *standard input* of another command (1.5). Certain commands such as the line printer daemon *p* do not place their results on the *standard output* but rather in more useful places such as on the line printer (2.3). Similarly the *write* command places its output on another user's terminal rather than its *standard output* (2.3). Commands also have a *diagnostic output* where they write their error messages. Normally these go to the terminal even if the *standard output* has been sent to a file or another command, but it is possible to direct error diagnostics along with *standard output* using a special metanotation (2.5).

pushd — The *pushd* command, which means 'push directory', changes the shell's *working directory* and also remembers the current *working directory* before the change is made, allowing you to return to the same directory via the *popd* command later without retyping its name (2.7).

path — The shell has a variable *path* which gives the names of the directories in which it searches for the commands which it is given. It always checks first to see if the command it is given is built into the shell. If it is, then it need not search for the command as it can do it internally. If the command is not builtin, then the shell searches for a file with the name given in each of the directories in the *path* variable, left to right. Since the normal definition of the *path* variable is

path (. /usr/ucb /bin /usr/bin)

the shell normally looks in the current directory, and then in the standard system directories '/usr/ucb', '/bin' and '/usr/bin' for the named command (2.2). If the command cannot be found the shell will print an error diagnostic. Scripts of shell commands will be executed using another shell to interpret them if they have 'execute' permission set. This is normally true because a command of the form

chmod 755 script

was executed to turn this execute permission on (3.3). If you add new commands to a directory in the *path*, you should issue the command *rehash* (2.2).

pathname — A list of names, separated by '/' characters, forms a *pathname*. Each *component*, between successive '/' characters, names a directory in which the next *component* file resides. *Pathnames* which begin with the character '/' are interpreted relative to the *root* directory in the filesystem. Other *pathnames* are interpreted relative to the current directory as reported by *pwd*. The last component of a *pathname* may name a directory, but usually names a file.

pipeline — A group of commands which are connected together, the *standard output* of each connected to the *standard input* of the next, is called a *pipeline*. The *pipe* mechanism used to connect these commands is indicated by the shell metacharacter '|' (1.5, 2.3).

popd — The *popd* command changes the shell's *working directory* to the directory you most recently left using the *pushd* command. It returns to the directory without having to type its name, forgetting the name of the current *working directory* before doing so (2.7).

port — The part of a computer system to which each terminal is connected is called a *port*. Usually the system has a fixed number of *ports*, some of which are connected to telephone lines for dial-up access, and some of which are permanently wired directly to specific terminals.

pr — The *pr* command is used to prepare listings of the contents of files with headers giving the name of the file and the date and time at which the file was last modified (2.3).

printenv — The *printenv* command is used to print the current setting of variables in the environment (2.8).

process — An instance of a running program is called a *process* (2.6). UNIX assigns each *process* a unique number when it is started — called the *process number*. *Process numbers* can be used to stop individual *processes* using the *kill* or *stop* commands when the *processes* are part of a detached *background* job.

program — Usually synonymous with *command*; a binary file or shell command script which performs a useful function is often called a *program*.

programmer's manual — Also referred to as the *manual*. See the glossary entry for 'manual'.

prompt — Many programs will print a *prompt* on the terminal when they expect input. Thus the editor 'ex(1)' will print a ':' when it expects input. The shell *prompts* for input with '% ' and occasionally with '? ' when reading commands from the terminal (1.1). The shell has a variable *prompt* which may be set to a different value to change the shell's main *prompt*. This is mostly used when debugging the shell (2.8).

ps — The *ps* command is used to show the processes you are currently running. Each process is shown with its unique process number, an indication of the terminal name it is attached to, an indication of the state of the process (whether it is running, stopped, awaiting some event (sleeping), and whether it is swapped out), and the amount of CPU time it has used so far. The command is identified by printing some of the words used when it was invoked (2.6). Shells, such as the *csh* you use to run the *ps* command, are not normally shown in the output.

pwd — The *pwd* command prints the full *pathname* of the current *working directory*. The *dirs* builtin command is usually a better and faster choice.

quit — The *quit* signal, generated by a control-\, is used to terminate programs which are behaving unreasonably. It normally produces a core image file (1.8).

quotation — The process by which metacharacters are prevented their special meaning, usually by using the character ‘' in pairs, or by using the character ‘\’, is referred to as *quotation* (1.7).

redirection — The routing of input or output from or to a file is known as *redirection* of input or output (1.3).

rehash — The *rehash* command tells the shell to rebuild its internal table of which commands are found in which directories in your *path*. This is necessary when a new program is installed in one of these directories (2.8).

relative pathname — A *pathname* which does not begin with a ‘/’ is called a *relative pathname* since it is interpreted *relative* to the current *working directory*. The first *component* of such a *pathname* refers to some file or directory in the *working directory*, and subsequent *components* between ‘/’ characters refer to directories below the *working directory*. *Pathnames* that are not *relative* are called *absolute pathnames* (1.6).

repeat — The *repeat* command iterates another command a specified number of times.

root — The directory that is at the top of the entire directory structure is called the *root* directory since it is the ‘root’ of the entire tree structure of directories. The name used in *pathnames* to indicate the *root* is ‘/’. *Pathnames* starting with ‘/’ are said to be *absolute* since they start at the *root* directory. *Root* is also used as the part of a *pathname* that is left after removing the *extension*. See *filename* for a further explanation (1.6).

RUBOUT — The RUBOUT or DELETE key sends an interrupt to the current job. Most interactive commands return to their command level upon receipt of an interrupt, while non-interactive commands usually terminate, returning control to the shell. Users often change interrupt to be generated by ↑C rather than DELETE by using the *stty* command.

scratch file — Files whose names begin with a ‘#’ are referred to as *scratch files*, since they are automatically removed by the system after a couple of days of non-use, or more frequently if disk space becomes tight (1.3).

script — Sequences of shell commands placed in a file are called shell command *scripts*. It is often possible to perform simple tasks using these *scripts* without writing a program in a language such as C, by using the shell to selectively run other programs (3.3, 3.10).

set — The builtin *set* command is used to assign new values to shell variables and to show the values of the current variables. Many shell variables have special meaning to the shell itself. Thus by using the *set* command the behavior of the shell can be affected (2.1).

setenv — Variables in the environment 'environ(5)' can be changed by using the *setenv* builtin command (2.8). The *printenv* command can be used to print the value of the variables in the environment.

shell — A *shell* is a command language interpreter. It is possible to write and run your own *shell*, as *shells* are no different than any other programs as far as the system is concerned. This manual deals with the details of one particular *shell*, called *csh*.

shell script — See *script* (3.3, 3.10).

signal — A *signal* in UNIX is a short message that is sent to a running program which causes something to happen to that process. *Signals* are sent either by typing special *control* characters on the keyboard or by using the *kill* or *stop* commands (1.8, 2.6).

sort — The *sort* program sorts a sequence of lines in ways that can be controlled by argument *flags* (1.5).

source — The *source* command causes the shell to read commands from a specified file. It is most useful for reading files such as *.cshrc* after changing them (2.8).

special character — See *metacharacters* and the appendix to this manual.

standard — We refer often to the *standard input* and *standard output* of commands. See *input* and *output* (1.3, 3.8).

status — A command normally returns a *status* when it finishes. By convention a *status* of zero indicates that the command succeeded. Commands may return non-zero *status* to indicate that some abnormal event has occurred. The shell variable *status* is set to the *status* returned by the last command. It is most useful in shell commmand scripts (3.6).

stop — The *stop* command causes a *background* job to become *suspended* (2.6).

string — A sequential group of characters taken together is called a *string*. *Strings* can contain any printable characters (2.2).

stty — The *stty* program changes certain parameters inside UNIX which determine how your terminal is handled. See 'stty(1)' for a complete description (2.6).

substitution — The shell implements a number of *substitutions* where sequences indicated by metacharacters are replaced by other sequences. Notable examples of this are history *substitution* keyed by the metacharacter '!' and variable *substitution* indicated by '$'. We also refer to *substitutions* as *expansions* (3.4).

suspended — A job becomes *suspended* after a STOP signal is sent to it, either by typing a *control*-z at the terminal (for *foreground* jobs) or by using the *stop* command (for *background* jobs). When *suspended*, a job temporarily stops running until it is restarted by either the *fg* or *bg* command (2.6).

switch — The *switch* command of the shell allows the shell to select one of a number of sequences of commands based on an argument string. It is similar to the *switch* statement in the language C (3.7).

termination — When a command which is being executed finishes we say it undergoes *termination* or *terminates*. Commands normally terminate when they read an *end-of-file* from their *standard input*. It is also possible to terminate commands by sending them an *interrupt* or *quit* signal (1.8). The *kill* program terminates specified jobs (2.6).

then — The *then* command is part of the shell's 'if-then-else-endif' control construct used in command scripts (3.6).

time — The *time* command can be used to measure the amount of CPU and real time consumed by a specified command as well as the amount of disk i/o, memory utilized, and number of page faults and swaps taken by the command (2.1, 2.8).

tset — The *tset* program is used to set standard erase and kill characters and to tell the system what kind of terminal you are using. It is often invoked in a *.login* file (2.1).

tty — The word *tty* is a historical abbreviation for 'teletype' which is frequently used in UNIX to indicate the *port* to which a given terminal is connected. The *tty* command will print the name of the *tty* or *port* to which your terminal is presently connected.

unalias — The *unalias* command removes aliases (2.8).

UNIX — UNIX is an operating system on which *csh* runs. UNIX provides facilities which allow *csh* to invoke other programs such as editors and text formatters which you may wish to use.

unset — The *unset* command removes the definitions of shell variables (2.2, 2.8).

variable expansion
See *variables* and *expansion* (2.2, 3.4).

variables — *Variables* in *csh* hold one or more strings as value. The most common use of *variables* is in controlling the behavior of the shell. See *path*, *noclobber*, and *ignoreeof* for examples. *Variables* such as *argv* are also used in writing shell programs (shell command scripts) (2.2).

verbose — The *verbose* shell variable can be set to cause commands to be echoed after they are history expanded. This is often useful in debugging shell scripts. The *verbose* variable is set by the shell's *−v* command line option (3.10).

wc — The *wc* program calculates the number of characters, words, and lines in the files whose names are given as arguments (2.6).

while — The *while* builtin control construct is used in shell command scripts (3.7).

word — A sequence of characters which forms an argument to a command is called a *word*. Many characters which are neither letters, digits, '−', '.' nor '/' form *words* all by themselves even if they are not surrounded by blanks. Any sequence of characters may be made into a *word* by surrounding it with ''' characters except for the characters ''' and '!' which require special treatment (1.1). This process of placing special characters in *words* without their special meaning is called *quoting*.

working directory
At any given time you are in one particular directory, called your *working directory*. This directory's name is printed by the *pwd* command and the files listed by *ls* are the ones in this directory. You can change *working directories* using *chdir*.

write — The *write* command is used to communicate with other users who are logged in to UNIX.

A Guide to the Dungeons of Doom

Michael C. Toy
Kenneth C. R. C. Arnold

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

ABSTRACT

Rogue is a visual CRT based fantasy game which runs under the UNIX† timesharing system. This paper describes how to play rogue, and gives a few hints for those who might otherwise get lost in the Dungeons of Doom.

†UNIX is a trademark of Bell Laboratories

1. Introduction

You have just finished your years as a student at the local fighter's guild. After much practice and sweat you have finally completed your training and are ready to embark upon a perilous adventure. As a test of your skills, the local guildmasters have sent you into the Dungeons of Doom. Your task is to return with the Amulet of Yendor. Your reward for the completion of this task will be a full membership in the local guild. In addition, you are allowed to keep all the loot you bring back from the dungeons.

In preparation for your journey, you are given an enchanted mace, a bow, and a quiver of arrows taken from a dragon's hoard in the far off Dark Mountains. You are also outfitted with elf-crafted armor and given enough food to reach the dungeons. You say goodbye to family and friends for what may be the last time and head up the road.

You set out on your way to the dungeons and after several days of uneventful travel, you see the ancient ruins that mark the entrance to the Dungeons of Doom. It is late at night, so you make camp at the entrance and spend the night sleeping under the open skies. In the morning you gather your weapons, put on your armor, eat what is almost your last food, and enter the dungeons.

2. What is going on here?

You have just begun a game of rogue. Your goal is to grab as much treasure as you can, find the Amulet of Yendor, and get out of the Dungeons of Doom alive. On the screen, a map of where you have been and what you have seen on the current dungeon level is kept. As you explore more of the level, it appears on the screen in front of you.

Rogue differs from most computer fantasy games in that it is screen oriented. Commands are all one or two keystrokes[1] and the results of your commands are displayed graphically on the screen rather than being explained in words.[2]

Another major difference between rogue and other computer fantasy games is that once you have solved all the puzzles in a standard fantasy game, it has lost most of its excitement and it ceases to be fun. Rogue, on the other hand, generates a new dungeon every time you play it and even the author finds it an entertaining and exciting game.

3. What do all those things on the screen mean?

In order to understand what is going on in rogue you have to first get some grasp of what rogue is doing with the screen. The rogue screen is intended to replace the "You can see ..." descriptions of standard fantasy games. Figure 1 is a sample of what a rogue screen might look like.

3.1. The bottom line

At the bottom line of the screen are a few pieces of cryptic information describing your current status. Here is an explanation of what these things mean:

Level This number indicates how deep you have gone in the dungeon. It starts at one and goes up as you go deeper into the dungeon.

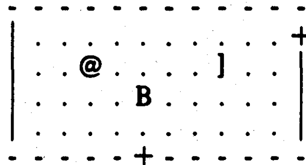
Gold The number of gold pieces you have managed to find and keep with you so far.

Hp Your current and maximum hit points. Hit points indicate how much damage you can take before you die. The more you get hit in a fight, the lower they get. You can regain hit points by resting. The number in parentheses is the maximum number your hit points can reach.

---

[1] As opposed to pseudo English sentences.

[2] A minimum screen size of 24 lines by 80 columns is required. If the screen is larger, only the 24x80 section will be used for the map.



Level: 1 Gold: 0 Hp: 12(12) Str: 16(16) Ac: 6 Exp: 1/0

Figure 1

Str Your current strength and maximum ever strength. This can be any integer less than or equal to 31, or greater than or equal to three. The higher the number, the stronger you are. The number in the parentheses is the maximum strength you have attained so far this game.

Ac Your current armor class. This number indicates how effective your armor is in stopping blows from unfriendly creatures. The lower this number is, the more effective the armor.

Exp These two numbers give your current experience level and experience points. As you do things, you gain experience points. At certain experience point totals, you gain an experience level. The more experienced you are, the better you are able to fight and to withstand magical attacks.

3.2. The top line

The top line of the screen is reserved for printing messages that describe things that are impossible to represent visually. If you see a "--More--" on the top line, this means that rogue wants to print another message on the screen, but it wants to make certain that you have read the one that is there first. To read the next message, just type a space.

3.3. The rest of the screen

The rest of the screen is the map of the level as you have explored it so far. Each symbol on the screen represents something. Here is a list of what the various symbols mean:

@ This symbol represents you, the adventurer.

-| These symbols represent the walls of rooms.

+ A door to/from a room.

. The floor of a room.

The floor of a passage between rooms.

* A pile or pot of gold.

) A weapon of some sort.

] A piece of armor.

! A flask containing a magic potion.

? A piece of paper, usually a magic scroll.

= A ring with magic properties

/ A magical staff or wand

^ A trap, watch out for these.

% A staircase to other levels

: A piece of food.

A-Z The uppercase letters represent the various inhabitants of the Dungeons of Doom. Watch out, they can be nasty and vicious.

4. Commands

Commands are given to rogue by typing one or two characters. Most commands can be preceded by a count to repeat them (e.g. typing "10s" will do ten searches). Commands for which counts make no sense have the count ignored. To cancel a count or a prefix, type <ESCAPE>. The list of commands is rather long, but it can be read at any time during the game with the "?" command. Here it is for reference, with a short explanation of each command.

? The help command. Asks for a character to give help on. If you type a "*", it will list all the commands, otherwise it will explain what the character you typed does.

/ This is the "What is that on the screen?" command. A "/" followed by any character that you see on the level, will tell you what that character is. For instance, typing "/@" will tell you that the "@" symbol represents you, the player.

h, H, ^H
Move left. You move one space to the left. If you use upper case "h", you will continue to move left until you run into something. This works for all movement commands (e.g. "L" means run in direction "l".) If you use the "control" "h", you will continue moving in the specified direction until you pass something interesting or run into a wall. You should experiment with this, since it is a very useful command, but very difficult to describe. This also works for all movement commands.

j Move down.

k Move up.

l Move right.

y Move diagonally up and left.

u Move diagonally up and right.

b Move diagonally down and left.

n Move diagonally down and right.

t Throw an object. This is a prefix command. When followed with a direction it throws an object in the specified direction. (e.g. type "th" to throw something to the left.)

f Fight until someone dies. When followed with a direction this will force you to fight the creature in that direction until either you or it bites the big one.

m Move onto something without picking it up. This will move you one space in the direction you specify and, if there is an object there you can pick up, it won't do it.

z Zap prefix. Point a staff or wand in a given direction and fire it. Even non-directional staves must be pointed in some direction to be used.

^ Identify trap command. If a trap is on your map and you can't remember what type it is, you can get rogue to remind you by getting next to it and typing "^" followed by the direction that would move you on top of it.

s Search for traps and secret doors. Examine each space immediately adjacent to you for the existence of a trap or secret door. There is a large chance that even if there is something there, you won't find it, so you might have to search a while before you find

something.

> Climb down a staircase to the next level. Not surprisingly, this can only be done if you are standing on staircase.

< Climb up a staircase to the level above. This can't be done without the Amulet of Yendor in your possession.

. Rest. This is the "do nothing" command. This is good for waiting and healing.

* Inventory. List what you are carrying in your pack.

I Selective inventory. Tells you what a single item in your pack is.

q Quaff one of the potions you are carrying.

r Read one of the scrolls in your pack.

e Eat food from your pack.

w Wield a weapon. Take a weapon out of your pack and carry it for use in combat, replacing the one you are currently using (if any).

W Wear armor. You can only wear one suit of armor at a time. This takes extra time.

T Take armor off. You can't remove armor that is cursed. This takes extra time.

P Put on a ring. You can wear only two rings at a time (one on each hand). If you aren't wearing any rings, this command will ask you which hand you want to wear it on, otherwise, it will place it on the unused hand. The program assumes that you wield your sword in your right hand.

R Remove a ring. If you are only wearing one ring, this command takes it off. If you are wearing two, it will ask you which one you wish to remove,

d Drop an object. Take something out of your pack and leave it lying on the floor. Only one object can occupy each space. You cannot drop a cursed object at all if you are wielding or wearing it.

c Call an object something. If you have a type of object in your pack which you wish to remember something about, you can use the call command to give a name to that type of object. This is usually used when you figure out what a potion, scroll, ring, or staff is after you pick it up, or when you want to remember which of those swords in your pack you were wielding.

D Print out which things you've discovered something about. This command will ask you what type of thing you are interested in. If you type the character for a given type of object (*e.g.* "!" for potion) it will tell you which kinds of that type of object you've discovered (*i.e.*, figured out what they are). This command works for potions, scrolls, rings, and staves and wands.

o Examine and set options. This command is further explained in the section on options.

^R Redraws the screen. Useful if spurious messages or transmission errors have messed up the display.

^P Print last message. Useful when a message disappears before you can read it. This only repeats the last message that was not a mistyped command so that you don't loose anything by accidentally typing the wrong character instead of ^P.

<ESCAPE>
Cancel a command, prefix, or count.

! Escape to a shell for some commands.

Q Quit. Leave the game.

S Save the current game in a file. It will ask you whether you wish to use the default save file. *Caveat*: Rogue won't let you start up a copy of a saved game, and it removes the save file as soon as you start up a restored game. This is to prevent people from saving a

game just before a dangerous position and then restarting it if they die. To restore a saved game, give the file name as an argument to rogue. As in

% rogue *save_file*

To restart from the default save file (see below), run

% rogue −r

v Prints the program version number.

) Print the weapon you are currently wielding

] Print the armor you are currently wearing

= Print the rings you are currently wearing

@ Reprint the status line on the message line

5. Rooms

Rooms in the dungeons are either lit or dark. If you walk into a lit room, the entire room will be drawn on the screen as soon as you enter. If you walk into a dark room, it will only be displayed as you explore it. Upon leaving a room, all monsters inside the room are erased from the screen. In the darkness you can only see one space in all directions around you. A corridor is always dark.

6. Fighting

If you see a monster and you wish to fight it, just attempt to run into it. Many times a monster you find will mind its own business unless you attack it. It is often the case that discretion is the better part of valor.

7. Objects you can find

When you find something in the dungeon, it is common to want to pick the object up. This is accomplished in rogue by walking over the object (unless you use the "m" prefix, see above). If you are carrying too many things, the program will tell you and it won't pick up the object, otherwise it will add it to your pack and tell you what you just picked up.

Many of the commands that operate on objects must prompt you to find out which object you want to use. If you change your mind and don't want to do that command after all, just type an <ESCAPE> and the command will be aborted.

Some objects, like armor and weapons, are easily differentiated. Others, like scrolls and potions, are given labels which vary according to type. During a game, any two of the same kind of object with the same label are the same type. However, the labels will vary from game to game.

When you use one of these labeled objects, if its effect is obvious, rogue will remember what it is for you. If it's effect isn't extremely obvious you will be asked what you want to scribble on it so you will recognize it later, or you can use the "call" command (see above).

7.1. Weapons

Some weapons, like arrows, come in bunches, but most come one at a time. In order to use a weapon, you must wield it. To fire an arrow out of a bow, you must first wield the bow, then throw the arrow. You can only wield one weapon at a time, but you can't change weapons if the one you are currently wielding is cursed. The commands to use weapons are "w" (wield) and "t" (throw).

7.2. Armor

There are various sorts of armor lying around in the dungeon. Some of it is enchanted, some is cursed, and some is just normal. Different armor types have different armor classes. The lower the armor class, the more protection the armor affords against the blows of

monsters. Here is a list of the various armor types and their normal armor class:

Type	*Class*
None	10
Leather armor	8
Studded leather / Ring mail	7
Scale mail	6
Chain mail	5
Banded mail / Splint mail	4
Plate mail	3

If a piece of armor is enchanted, its armor class will be lower than normal. If a suit of armor is cursed, its armor class will be higher, and you will not be able to remove it. However, not all armor with a class that is higher than normal is cursed.

The commands to use weapons are "W" (wear) and "T" (take off).

7.3. Scrolls

Scrolls come with titles in an unknown tongue[3]. After you read a scroll, it disappears from your pack. The command to use a scroll is "r" (read).

7.4. Potions

Potions are labeled by the color of the liquid inside the flask. They disappear after being quaffed. The command to use a scroll is "q" (quaff).

7.5. Staves and Wands

Staves and wands do the same kinds of things. Staves are identified by a type of wood; wands by a type of metal or bone. They are generally things you want to do to something over a long distance, so you must point them at what you wish to affect to use them. Some staves are not affected by the direction they are pointed, though. Staves come with multiple magic charges, the number being random, and when they are used up, the staff is just a piece of wood or metal.

The command to use a wand or staff is "z" (zap)

7.6. Rings

Rings are very useful items, since they are relatively permanent magic, unlike the usually fleeting effects of potions, scrolls, and staves. Of course, the bad rings are also more powerful. Most rings also cause you to use up food more rapidly, the rate varying with the type of ring. Rings are differentiated by their stone settings. The commands to use rings are "P" (put on) and "R" (remove).

7.7. Food

Food is necessary to keep you going. If you go too long without eating you will faint, and eventually die of starvation. The command to use food is "e" (eat).

8. Options

Due to variations in personal tastes and conceptions of the way rogue should do things, there are a set of options you can set that cause rogue to behave in various different ways.

---

[3] Actually, it's a dialect spoken only by the twenty-seven members of a tribe in Outer Mongolia, but you're not supposed to *know* that.

8.1. Setting the options

There are two ways to set the options. The first is with the "o" command of rogue; the second is with the "ROGUEOPTS" environment variable[4].

8.1.1. Using the 'o' command

When you type "o" in rogue, it clears the screen and displays the current settings for all the options. It then places the cursor by the value of the first option and waits for you to type. You can type a <RETURN> which means to go to the next option, a "−" which means to go to the previous option, an <ESCAPE> which means to return to the game, or you can give the option a value. For boolean options this merely involves typing "t" for true or "f" for false. For string options, type the new value followed by a <RETURN>.

8.1.2. Using the ROGUEOPTS variable

The ROGUEOPTS variable is a string containing a comma separated list of initial values for the various options. Boolean variables can be turned on by listing their name or turned off by putting a "no" in front of the name. Thus to set up an environment variable so that **jump** is on, **terse** is off, and the **name** is set to "Blue Meanie", use the command

% setenv ROGUEOPTS "jump,noterse,name=Blue Meanie"[5]

8.2. Option list

Here is a list of the options and an explanation of what each one is for. The default value for each is enclosed in square brackets. For character string options, input over fifty characters will be ignored.

terse [*noterse*]
: Useful for those who are tired of the sometimes lengthy messages of rogue. This is a useful option for playing on slow terminals, so this option defaults to *terse* if you are on a slow (1200 baud or under) terminal.

jump [*nojump*]
: If this option is set, running moves will not be displayed until you reach the end of the move. This saves considerable cpu and display time. This option defaults to *jump* if you are using a slow terminal.

flush [*noflush*]
: All typeahead is thrown away after each round of battle. This is useful for those who type far ahead and then watch in dismay as a Bat kills them.

seefloor [*seefloor*]
: Display the floor around you on the screen as you move through dark rooms. Due to the amount of characters generated, this option defaults to *noseefloor* if you are using a slow terminal.

passgo [*nopassgo*]
: Follow turnings in passageways. If you run in a passage and you run into stone or a wall, rogue will see if it can turn to the right or left. If it can only turn one way, it will turn that way. If it can turn either or neither, it will stop. This is followed strictly, which can sometimes lead to slightly confusing occurrences (which is why it defaults to *nopassgo*).

tombstone [*tombstone*]
: Print out the tombstone at the end if you get killed. This is nice but slow, so you can turn it off if you like.

---

[4] On Version 6 systems, there is no equivalent of the ROGUEOPTS feature.

[5] For those of you who use the bourne shell, the commands would be
$ ROGUEOPTS="jump,noterse,name=Blue Meanie"
$ export ROGUEOPTS

inven [*overwrite*]
Inventory type. This can have one of three values: *overwrite*, *slow*, or *clear*. With *overwrite* the top lines of the map are overwritten with the list when inventory is requested or when "Which item do you wish to . . . ? " questions are answered with a "*". However, if the list is longer than a screenful, the screen is cleared. With *slow*, lists are displayed one item at a time on the top of the screen, and with *clear*, the screen is cleared, the list is displayed, and then the dungeon level is re-displayed. Due to speed considerations, *clear* is the default for terminals without clear-to-end-of-line capabilities.

name [account name]
This is the name of your character. It is used if you get on the top ten scorer's list.

fruit [*slime-mold*]
This should hold the name of a fruit that you enjoy eating. It is basically a whimsey that rogue uses in a couple of places.

file [*~/rogue.save*]
The default file name for saving the game. If your phone is hung up by accident, rogue will automatically save the game in this file. The file name may start with the special character "~" which expands to be your home directory.

9. Scoring

Rogue usually maintains a list of the top scoring people or scores on your machine. Depending on how it is set up, it can post either the top scores or the top players. In the latter case, each account on the machine can post only one non-winning score on this list. If you score higher than someone else on this list, or better your previous score on the list, you will be inserted in the proper place under your current name. How many scores are kept can also be set up by whoever installs it on your machine.

If you quit the game, you get out with all of your gold intact. If, however, you get killed in the Dungeons of Doom, your body is forwarded to your next-of-kin, along with 90% of your gold; ten percent of your gold is kept by the Dungeons' wizard as a fee[6]. This should make you consider whether you want to take one last hit at that monster and possibly live, or quit and thus stop with whatever you have. If you quit, you do get all your gold, but if you swing and live, you might find more.

If you just want to see what the current top players/games list is, you can type

```
% rogue −s
```

10. Acknowledgements

Rogue was originally conceived of by Glenn Wichman and Michael Toy. Ken Arnold and Michael Toy then smoothed out the user interface, and added jillions of new features. We would like to thank Bob Arnold, Michelle Busch, Andy Hatcher, Kipp Hickman, Mark Horton, Daniel Jensen, Bill Joy, Joe Kalash, Steve Maurer, Marty McNary, Jan Miller, and Scott Nelson for their ideas and assistance; and also the teeming multitudes who graciously ignored work, school, and social life to play rogue and send us bugs, complaints, suggestions, and just plain flames. And also Mom.

[6] The Dungeon's wizard is named Wally the Wonder Badger. Invocations should be accompanied by a sizable donative.

Screen Updating and Cursor Movement Optimization: A Library Package

Kenneth C. R. C. Arnold

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

ABSTRACT

This document describes a package of C library functions which allow the user to:

- update a screen with reasonable optimization,
- get input from the terminal in a screen-oriented fashion, and
- independent from the above, move the cursor optimally from one point to another.

These routines all use the **/etc/termcap** database to describe the capabilities of the terminal.

Acknowledgements

This package would not exist without the work of Bill Joy, who, in writing his editor, created the capability to generally describe terminals, wrote the routines which read this database, and, most importantly, those which implement optimal cursor movement, which routines I have simply lifted nearly intact. Doug Merritt and Kurt Shoens also were extremely important, as were both willing to waste time listening to me rant and rave. The help and/or support of Ken Abrams, Alan Char, Mark Horton, and Joe Kalash, was, and is, also greatly appreciated.

Screen Package

Contents

1 Overview 1
  1.1 Terminology (or, Words You Can Say to Sound Brilliant) 1
  1.2 Compiling Things 1
  1.3 Screen Updating 1
  1.4 Naming Conventions 2
2 Variables 2
3 Usage 3
  3.1 Starting up 3
  3.2 The Nitty-Gritty 3
    3.2.1 Output 3
    3.2.2 Input 4
    3.2.3 Miscellaneous 4
  3.3 Finishing up 4
4 Cursor Motion Optimization: Standing Alone 4
  4.1 Terminal Information 4
  4.2 Movement Optimizations, or, Getting Over Yonder 5
5 The Functions 6
  5.1 Output Functions 6
  5.2 Input Functions 9
  5.3 Miscellaneous Functions 10
  5.4 Details 13

Appendixes

Appendix A 14
1 Capabilities from termcap 14
  1.1 Disclaimer 14
  1.2 Overview 14
  1.3 Variables Set By setterm() 14
  1.4 Variables Set By gettmode() 15
Appendix B 16
1 The WINDOW structure 16
Appendix C 17
1 Examples 17
2 Screen Updating 17
  2.1 Twinkle 17
  2.2 Life 19
3 Motion optimization 22
  3.1 Twinkle 22

Screen Package

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 get 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 *getttmode()* 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. This returns ERR if it would cause the screen to scroll illegally.

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. This returns ERR if it would cause the screen to scroll illegally.

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.

getch() †

wgetch(win)
WINDOW **win;*

Gets a character from the terminal and (if necessary) echos it on the window. This returns ERR if it would cause the screen to scroll illegally. Otherwise, the character gotten is returned. If *noecho* has been set, then the window is left unaltered. In order to retain control of the terminal, it is necessary to have one of *noecho*, *cbreak*, or *rawmode* set. If you do not set one, whatever routine you call to read characters will set *cbreak* for you, and then reset to the original mode when finished.

getstr(str) †
char **str;*

wgetstr(win, str)
WINDOW **win;*
char **str;*

Get a string through the window and put it in the location pointed to by *str*, which is assumed to be large enough to handle it. It sets tty modes if necessary, and then calls *getch()* (or *wgetch(win)*) to get the characters needed to fill in the string until a newline or EOF is encountered. The newline stripped off the string. This returns ERR if it would cause the screen to scroll illegally.

raw() †

noraw() †

Set or unset the terminal to/from raw mode. On version 7 UNIX[9] this also turns of newline mapping (see *nl()*).

scanw(fmt, arg1, arg2, ...)
char **fmt;*

wscanw(win, fmt, arg1, arg2, ...)
WINDOW **win;*
char **fmt;*

Perform a *scanf()* through the window using *fmt*. It does this using consecutive *getch()*'s (or *wgetch(win)*'s). This returns ERR if it would cause the screen to scroll illegally.

5.3. Miscellaneous Functions

[9] UNIX is a trademark of Bell Laboratories.

delwin(win)
*WINDOW *win;*

Deletes the window from existence. All resources are freed for future use by **calloc**(3). If a window has a *subwin()* allocated window inside of it, deleting the outer window the subwindow is not affected, even though this does invalidate it. Therefore, subwindows should be deleted before their outer windows are.

endwin()

Finish up window routines before exit. This restores the terminal to the state it was before *initscr()* (or *gettmode()* and *setterm()*) was called. It should always be called before exiting. It does not exit. This is especially useful for resetting tty stats when trapping rubouts via **signal**(2).

getyx(win, y, x) †
*WINDOW *win;*
int y, x;

Puts the current (y, x) co-ordinates of *win* in the variables *y* and *x*. Since it is a macro, not a function, you do not pass the address of *y* and *x*.

inch() †

winch(win) †
*WINDOW *win;*

Returns the character at the current (y, x) co-ordinates on the given window. This does not make any changes to the window. This has no associated "**mv**" command.

initscr()

Initialize the screen routines. This must be called before any of the screen routines are used. It initializes the terminal-type data and such, and without it, none of the routines can operate. If standard input is not a tty, it sets the specifications to the terminal whose name is pointed to by *Def_term* (initialy "dumb"). If the boolean *My_term* is true, *Def_term* is always used.

leaveok(win, boolf) †
*WINDOW *win;*
bool boolf;

Sets the boolean flag for leaving the cursor after the last change. If *boolf* is TRUE, the cursor will be left after the last update on the terminal, and the current (y, x) co-ordinates for *win* will be changed accordingly. If it is FALSE, it will be moved to the current (y, x) co-ordinates. This flag (initialy FALSE) retains its value until changed by the user.

longname(termbuf, name)
*char *termbuf, *name;*

Fills in *name* with the long (full) name of the terminal described by the termcap entry in *termbuf*. It is generally of little use, but is nice for telling the user in a readable format what terminal we think he has. This is available in the global variable *ttytype*. *Termbuf* is usually set via the termlib routine *tgetent()*.

mvwin(win, y, x)
*WINDOW *win;*
int y, x;

Move the home position of the window *win* from its current starting coordinates to (*y*, *x*). If that would put part or all of the window off the edge of the terminal screen, *mvwin()* returns ERR and does not change anything.

*WINDOW **
newwin(lines, cols, begin_y, begin_x)
int lines, cols, begin_y, begin_x;

Create a new window with *lines* lines and *cols* columns starting at position (*begin_y*, *begin_x*). If either *lines* or *cols* is 0 (zero), that dimension will be set to (*LINES* − *begin_y*) or (*COLS* − *begin_x*) respectively. Thus, to get a new window of dimensions *LINES* × *COLS*, use *newwin(0, 0, 0, 0)*.

nl() †

nonl() †

Set or unset the terminal to/from nl mode, i.e., start/stop the system from mapping **<RETURN>** to **<LINE-FEED>**. If the mapping is not done, *refresh()* can do more optimization, so it is recommended, but not required, to turn it off.

scrollok(win, boolf) †
*WINDOW *win;*
bool boolf;

Set the scroll flag for the given window. If *boolf* is FALSE, scrolling is not allowed. This is its default setting.

touchwin(win)
*WINDOW *win;*

Make it appear that the every location on the window has been changed. This is usually only needed for refreshes with overlapping windows.

*WINDOW **
subwin(win, lines, cols, begin_y, begin_x)
*WINDOW *win;*
int lines, cols, begin_y, begin_x;

Create a new window with *lines* lines and *cols* columns starting at position (*begin_y*, *begin_x*) in the middle of the window *win*. This means that any change made to either window in the area covered by the subwindow will be made on both windows. *begin_y*, *begin_x* are specified relative to the overall screen, not the relative (0, 0) of *win*. If either *lines* or *cols* is 0 (zero), that dimension will be set to (*LINES* − *begin_y*) or

(*COLS* − *begin_x*) respectively.

unctrl(ch) †
char ch;

This is actually a debug function for the library, but it is of general usefulness. It returns a string which is a representation of *ch*. Control characters become their upper-case equivalents preceded by a "^". Other letters stay just as they are. To use *unctrl()*, you must have **#include <unctrl.h>** in your file.

5.4. Details

gettmode()

Get the tty stats. This is normally called by *initscr()*.

mvcur(lasty, lastx, newy, newx)
int lasty, lastx, newy, newx;

Moves the terminal's cursor from (*lasty, lastx*) to (*newy, newx*) in an approximation of optimal fashion. This routine uses the functions borrowed from *ex* version 2.6. It is possible to use this optimization without the benefit of the screen routines. With the screen routines, this should not be called by the user. *move()* and *refresh()* should be used to move the cursor position, so that the routines know what's going on.

scroll(win)
*WINDOW *win;*

Scroll the window upward one line. This is normally not used by the user.

savetty() †

resetty() †

savetty() saves the current tty characteristic flags. *resetty()* restores them to what *savetty()* stored. These functions are performed automatically by *initscr()* and *endwin()*.

setterm(name)
*char *name;*

Set the terminal characteristics to be those of the terminal named *name*. This is normally called by *initscr()*.

tstp()

If the new **tty**(4) driver is in use, this function will save the current tty state and then put the process to sleep. When the process gets restarted, it restores the tty state and then calls *wrefresh(curscr)* to redraw the screen. *initscr()* sets the signal SIGTSTP to trap to this routine.

Appendix A

1. Capabilities from termcap

1.1. Disclaimer

The description of terminals is a difficult business, and we only attempt to summarize the capabilities here: for a full description see the paper describing termcap.

1.2. Overview

Capabilities from termcap are of three kinds: string valued options, numeric valued options, and boolean options. The string valued options are the most complicated, since they may include padding information, which we describe now.

Intelligent terminals often require padding on intelligent operations at high (and sometimes even low) speed. This is specified by a number before the string in the capability, and has meaning for the capabilities which have a **P** at the front of their comment. This normally is a number of milliseconds to pad the operation. In the current system which has no true programmable delays, we do this by sending a sequence of pad characters (normally nulls, but can be changed (specified by *PC*)). In some cases, the pad is better computed as some number of milliseconds times the number of affected lines (to the bottom of the screen usually, except when terminals have insert modes which will shift several lines.) This is specified as, e.g., **12*.** before the capability, to say 12 milliseconds per affected whatever (currently always line). Capabilities where this makes sense say **P***.

1.3. Variables Set By setterm()

variables set by *setterm()*

Type	Name	Pad	Description
char *	AL	P*	Add new blank Line
bool	AM		Automatic Margins
char *	BC		Back Cursor movement
bool	BS		BackSpace works
char *	BT	P	Back Tab
bool	CA		Cursor Addressable
char *	CD	P*	Clear to end of Display
char *	CE	P	Clear to End of line
char *	CL	P*	CLear screen
char *	CM	P	Cursor Motion
char *	DC	P*	Delete Character
char *	DL	P*	Delete Line sequence
char *	DM		Delete Mode (enter)
char *	DO		DOwn line sequence
char *	ED		End Delete mode
bool	EO		can Erase Overstrikes with ' '
char *	EI		End Insert mode
char *	HO		HOme cursor
bool	HZ		HaZeltine ~ braindamage
char *	IC	P	Insert Character
bool	IN		Insert-Null blessing
char *	IM		enter Insert Mode (IC usually set, too)
char *	IP	P*	Pad after char Inserted using IM+IE
char *	LL		quick to Last Line, column 0
char *	MA		ctrl character MAp for cmd mode
bool	MI		can Move in Insert mode
bool	NC		No Cr: \r sends \r\n then eats \n

variables set by *setterm()*

Type	Name	Pad	Description
char *	ND		Non-Destructive space
bool	OS		OverStrike works
char	PC		Pad Character
char *	SE		Standout End (may leave space)
char *	SF	P	Scroll Forwards
char *	SO		Stand Out begin (may leave space)
char *	SR	P	Scroll in Reverse
char *	TA	P	TAb (not ^I or with padding)
char *	TE		Terminal address enable Ending sequence
char *	TI		Terminal address enable Initialization
char *	UC		Underline a single Character
char *	UE		Underline Ending sequence
bool	UL		UnderLining works even though !OS
char *	UP		UPline
char *	US		Underline Starting sequence[10]
char *	VB		Visible Bell
char *	VE		Visual End sequence
char *	VS		Visual Start sequence
bool	XN		a Newline gets eaten after wrap

Names starting with ***X*** are reserved for severely nauseous glitches

1.4. Variables Set By gettmode()

variables set by *gettmode()*

type	name	description
bool	NONL	Term can't hack linefeeds doing a CR
bool	GT	Gtty indicates Tabs
bool	UPPERCASE	Terminal generates only uppercase letters

[10] US and UE, if they do not exist in the termcap entry, are copied from SO and SE in *setterm()*

Appendix B

1. The WINDOW structure

The WINDOW structure is defined as follows:

```
# define        WINDOW struct _win_st

struct _win_st {
        short   _cury, _curx;
        short   _maxy, _maxx;
        short   _begy, _begx;
        short   _flags;
        bool    _clear;
        bool    _leave;
        bool    _scroll;
        char    **_y;
        short   *_firstch;
        short   *_lastch;
};

# define        _SUBWIN         01
# define        _ENDLINE        02
# define        _FULLWIN        04
# define        _SCROLLWIN      010
# define        _STANDOUT       0200
```

_cury and *_curx* are the current (y, x) co-ordinates for the window. New characters added to the screen are added at this point. *_maxy* and *_maxx* are the maximum values allowed for (*_cury, _curx*). *_begy* and *_begx* are the starting (y, x) co-ordinates on the terminal for the window, i.e., the window's home. *_cury*, *_curx*, *_maxy*, and *_maxx* are measured relative to (*_begy, _begx*), not the terminal's home.

_clear tells if a clear-screen sequence is to be generated on the next *refresh()* call. This is only meaningful for screens. The initial clear-screen for the first *refresh()* call is generated by initially setting clear to be TRUE for *curscr*, which always generates a clear-screen if set, irrelevant of the dimensions of the window involved. *_leave* is TRUE if the current (y, x) co-ordinates and the cursor are to be left after the last character changed on the terminal, or not moved if there is no change. *_scroll* is TRUE if scrolling is allowed.

_y is a pointer to an array of lines which describe the terminal. Thus:

```
_y[i]
```

is a pointer to the *i*th line, and

```
_y[i][j]
```

is the *j*th character on the *i*th line.

_flags can have one or more values or'd into it. **_SUBWIN** means that the window is a subwindow, which indicates to *delwin()* that the space for the lines is not to be freed. **_ENDLINE** says that the end of the line for this window is also the end of a screen. **_FULLWIN** says that this window is a screen. **_SCROLLWIN** indicates that the last character of this screen is at the lower right-hand corner of the terminal; *i.e.*, if a character was put there, the terminal would scroll. **_STANDOUT** says that all characters added to the screen are in standout mode.

---

[11] All variables not normally accessed directly by the user are named with an initial "_" to avoid conflicts with the user's variables.

Appendix C

1. Examples

Here we present a few examples of how to use the package. They attempt to be representative, though not comprehensive.

2. Screen Updating

The following examples are intended to demonstrate the basic structure of a program using the screen updating sections of the package. Several of the programs require calculational sections which are irrelevant of to the example, and are therefore usually not included. It is hoped that the data structure definitions give enough of an idea to allow understanding of what the relevant portions do. The rest is left as an exercise to the reader, and will not be on the final.

2.1. Twinkle

This is a moderately simple program which prints pretty patterns on the screen that might even hold your interest for 30 seconds or more. It switches between patterns of asterisks, putting them on one by one in random order, and then taking them off in the same fashion. It is more efficient to write this using only the motion optimization, as is demonstrated below.

```
# include        <curses.h>
# include        <signal.h>

/*
 * the idea for this program was a product of the imagination of
 * Kurt Schoens.  Not responsible for minds lost or stolen.
 */

# define         NCOLS   80
# define         NLINES  24
# define         MAXPATTERNS     4

struct locs {
        char     y, x;
};

typedef struct locs             LOCS;

LOCS    Layout[NCOLS * NLINES];         /* current board layout */

int     Pattern,                        /* current pattern number */
        Numstars;                       /* number of stars in pattern */

main() {

        char            *getenv();
        int             die();

        srand(getpid());                        /* initialize random sequence */

        initscr();
        signal(SIGINT, die);
        noecho();
        nonl();
        leaveok(stdscr, TRUE);
        scrollok(stdscr, FALSE);
```

```
        for (;;) {
                makeboard();                    /* make the board setup */
                puton('*');                     /* put on '*'s */
                puton(' ');                     /* cover up with ' 's */
        }
}

/*
 * On program exit, move the cursor to the lower left corner by
 * direct addressing, since current location is not guaranteed.
 * We lie and say we used to be at the upper right corner to guarantee
 * absolute addressing.
 */
die() {

        signal(SIGINT, SIG_IGN);
        mvcur(0, COLS-1, LINES-1, 0);
        endwin();
        exit(0);
}


/*
 * Make the current board setup.  It picks a random pattern and
 * calls ison() to determine if the character is on that pattern
 * or not.
 */
makeboard() {

        reg int         y, x;
        reg LOCS        *lp;

        Pattern = rand() % MAXPATTERNS;
        lp = Layout;
        for (y = 0; y < NLINES; y++)
                for (x = 0; x < NCOLS; x++)
                        if (ison(y, x)) {
                                lp->y = y;
                                lp++->x = x;
                        }
        Numstars = lp - Layout;
}

/*
 * Return TRUE if (y, x) is on the current pattern.
 */
ison(y, x)
reg int   y, x; {

        switch (Pattern) {
          case 0:               /* alternating lines */
                return !(y & 01);
```

```
                case 1:                 /* box */
                        if (x >= LINES && y >= NCOLS)
                                return FALSE;
                        if (y < 3 | y >= NLINES - 3)
                                return TRUE;
                        return (x < 3 | x >= NCOLS - 3);
                case 2:                 /* holy pattern! */
                        return ((x + y) & 01);
                case 3:                 /* bar across center */
                        return (y >= 9 && y <= 15);
                }
                /* NOTREACHED */
}

puton(ch)
reg char                ch; {

                reg LOCS                *lp;
                reg int                 r;
                reg LOCS                *end;
                LOCS                    temp;

                end = &Layout[Numstars];
                for (lp = Layout; lp < end; lp++) {
                        r = rand() % Numstars;
                        temp = *lp;
                        *lp = Layout[r];
                        Layout[r] = temp;
                }

                for (lp = Layout; lp < end; lp++) {
                        mvaddch(lp->y, lp->x, ch);
                        refresh();
                }
}
```

2.2. Life

This program plays the famous computer pattern game of life (Scientific American, May, 1974). The calculational routines create a linked list of structures defining where each piece is. Nothing here claims to be optimal, merely demonstrative. This program, however, is a very good place to use the screen updating routines, as it allows them to worry about what the last position looked like, so you don't have to. It also demonstrates some of the input routines.

```
# include               <curses.h>
# include               <signal.h>

/*
 *              Run a life game. This is a demonstration program for
 * the Screen Updating section of the -lcurses cursor package.
 */

struct lst_st {                                 /* linked list element */
```

```
	int		y, x;		/* (y, x) position of piece */
	struct lst_st	*next, *last;	/* doubly linked */
};

typedef struct lst_st	LIST;

LIST	*Head;			/* head of linked list */

main(ac, av)
int	ac;
char	*av[]; {

	int	die();

	evalargs(ac, av);		/* evaluate arguments */

	initscr();			/* initialize screen package */
	signal(SIGINT, die);		/* set to restore tty stats */
	crmode();			/* set for char-by-char */
	noecho();			/*
	nonl();				/* for optimization */

	getstart();			/* get starting position */
	for (;;) {
		prboard();		/* print out current board */
		update();		/* update board position */
	}
}

/*
 * This is the routine which is called when rubout is hit.
 * It resets the tty stats to their original values.  This
 * is the normal way of leaving the program.
 */
die() {

	signal(SIGINT, SIG_IGN);		/* ignore rubouts */
	mvcur(0, COLS-1, LINES-1, 0);		/* go to bottom of screen */
	endwin();				/* set terminal to initial state */
	exit(0);
}

/*
 * Get the starting position from the user.  They keys u, i, o, j, l,
 * m, ,, and . are used for moving their relative directions from the
 * k key.  Thus, u move diagonally up to the left, , moves directly down,
 * etc.  x places a piece at the current position, " " takes it away.
 * The input can also be from a file.  The list is built after the
 * board setup is ready.
 */
getstart() {

	reg char	c;
	reg int		x, y;
```

```
		box(stdscr, '|', '_');			/* box in the screen */
		move(1, 1);				/* move to upper left corner */

		do {
			refresh();			/* print current position */
			if ((c=getch()) == 'q')
				break;
			switch (c) {
			  case 'u':
			  case 'i':
			  case 'o':
			  case 'j':
			  case 'l':
			  case 'm':
			  case ',':
			  case '.':
				adjustyx(c);
				break;
			  case 'f':
				mvaddstr(0, 0, "File name: ");
				getstr(buf);
				readfile(buf);
				break;
			  case 'x':
				addch('X');
				break;
			  case ' ':
				addch(' ');
				break;
			}
		}

		if (Head != NULL)				/* start new list */
			dellist(Head);
		Head = malloc(sizeof (LIST));

		/*
		 * loop through the screen looking for 'x's, and add a list
		 * element for each one
		 */
		for (y = 1; y < LINES - 1; y++)
			for (x = 1; x < COLS - 1; x++) {
				move(y, x);
				if (inch() == 'x')
					addlist(y, x);
			}
}

/*
 * Print out the current board position from the linked list
 */
prboard() {

		reg LIST		*hp;
```

```
		erase();				/* clear out last position */
		box(stdscr, '|', '_');			/* box in the screen */

		/*
		 * go through the list adding each piece to the newly
		 * blank board
		 */
		for (hp = Head; hp; hp = hp->next)
			mvaddch(hp->y, hp->x, 'X');

		refresh();
}
```

3. Motion optimization

The following example shows how motion optimization is written on its own. Programs which flit from one place to another without regard for what is already there usually do not need the overhead of both space and time associated with screen updating. They should instead use motion optimization.

3.1. Twinkle

The **twinkle** program is a good candidate for simple motion optimization. Here is how it could be written (only the routines that have been changed are shown):

```
main() {

		reg char		*sp;
		char			*getenv();
		int			_putchar(), die();

		srand(getpid());			/* initialize random sequence */

		if (isatty(0)) {
			gettmode();
			if (sp=getenv("TERM"))
				setterm(sp);
			signal(SIGINT, die);
		}
		else {
			printf("Need a terminal on %d\n", _tty_ch);
			exit(1);
		}
		_puts(TI);
		_puts(VS);

		noecho();
		nonl();
		tputs(CL, NLINES, _putchar);
		for (;;) {
			makeboard();			/* make the board setup */
			puton('*');			/* put on '*'s */
			puton(' ');			/* cover up with ' 's */
		}
}
```

```
/*
 * _putchar defined for tputs() (and _puts())
 */
_putchar(c)
reg char		c; {

		putchar(c);
}

puton(ch)
char		ch; {

		static int		lasty, lastx;
		reg LOCS		*lp;
		reg int			r;
		reg LOCS		*end;
		LOCS			temp;

		end = &Layout[Numstars];
		for (lp = Layout; lp < end; lp++) {
			r = rand() % Numstars;
			temp = *lp;
			*lp = Layout[r];
			Layout[r] = temp;
		}

		for (lp = Layout; lp < end; lp++)
				/* prevent scrolling */
			if (!AM | (lp->y < NLINES - 1 | lp->x < NCOLS - 1)) {
				mvcur(lasty, lastx, lp->y, lp->x);
				putchar(ch);
				lasty = lp->y;
				if ((lastx = lp->x + 1) >= NCOLS)
					if (AM) {
						lastx = 0;
						lasty++;
					}
					else
						lastx = NCOLS - 1;
			}
}
```

4.2BSD System Manual

Revised July, 1983

William Joy, Eric Cooper, Robert Fabry,
Samuel Leffler, Kirk McKusick and David Mosher

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

(415) 642-7780

ABSTRACT

This document summarizes the facilities provided by the 4.2BSD version of the UNIX operating system. It does not attempt to act as a tutorial for use of the system nor does it attempt to explain or justify the design of the system facilities. It gives neither motivation nor implementation details, in favor of brevity.

The first section describes the basic kernel functions provided to a UNIX process: process naming and protection, memory management, software interrupts, object references (descriptors), time and statistics functions, and resource controls. These facilities, as well as facilities for bootstrap, shutdown and process accounting, are provided solely by the kernel.

The second section describes the standard system abstractions for files and file systems, communication, terminal handling, and process control and debugging. These facilities are implemented by the operating system or by network server processes.

* UNIX is a trademark of Bell Laboratories.

TABLE OF CONTENTS

Introduction.

0. Notation and types

1. Kernel primitives

1.1. Processes and protection
- .1. Host and process identifiers
- .2. Process creation and termination
- .3. User and group ids
- .4. Process groups

1.2. Memory management
- .1. Text, data and stack
- .2. Mapping pages
- .3. Page protection control
- .4. Giving and getting advice

1.3. Signals
- .1. Overview
- .2. Signal types
- .3. Signal handlers
- .4. Sending signals
- .5. Protecting critical sections
- .6. Signal stacks

1.4. Timing and statistics
- .1. Real time
- .2. Interval time

1.5. Descriptors
- .1. The reference table
- .2. Descriptor properties
- .3. Managing descriptor references
- .4. Multiplexing requests
- .5. Descriptor wrapping

1.6. Resource controls
- .1. Process priorities
- .2. Resource utilization
- .3. Resource limits

1.7. System operation support
- .1. Bootstrap operations
- .2. Shutdown operations
- .3. Accounting

2. System facilities

2.1. Generic operations
.1. Read and write
.2. Input/output control
.3. Non-blocking and asynchronous operations

2.2. File system
.1 Overview
.2. Naming
.3. Creation and removal
.3.1. Directory creation and removal
.3.2. File creation
.3.3. Creating references to devices
.3.4. Portal creation
.3.6. File, device, and portal removal
.4. Reading and modifying file attributes
.5. Links and renaming
.6. Extension and truncation
.7. Checking accessibility
.8. Locking
.9. Disc quotas

2.3. Inteprocess communication
.1. Interprocess communication primitives
.1.1. Communication domains
.1.2. Socket types and protocols
.1.3. Socket creation, naming and service establishment
.1.4. Accepting connections
.1.5. Making connections
.1.6. Sending and receiving data
.1.7. Scatter/gather and exchanging access rights
.1.8. Using read and write with sockets
.1.9. Shutting down halves of full-duplex connections
.1.10. Socket and protocol options
.2. UNIX domain
.2.1. Types of sockets
.2.2. Naming
.2.3. Access rights transmission
.3. INTERNET domain
.3.1. Socket types and protocols
.3.2. Socket naming
.3.3. Access rights transmission
.3.4. Raw access

2.4. Terminals and devices
.1. Terminals
.1.1. Terminal input
.1.1.1 Input modes
.1.1.2 Interrupt characters
.1.1.3 Line editing
.1.2. Terminal output
.1.3. Terminal control operations
.1.4. Terminal hardware support
.2. Structured devices

.3. Unstructured devices

2.5. Process control and debugging

I. Summary of facilities

0. Notation and types

The notation used to describe system calls is a variant of a C language call, consisting of a prototype call followed by declaration of parameters and results. An additional keyword **result**, not part of the normal C language, is used to indicate which of the declared entities receive results. As an example, consider the *read* call, as described in section 2.1:

```
cc = read(fd, buf, nbytes);
result int cc; int fd; result char *buf; int nbytes;
```

The first line shows how the *read* routine is called, with three parameters. As shown on the second line *cc* is an integer and *read* also returns information in the parameter *buf.*

Description of all error conditions arising from each system call is not provided here; they appear in the programmer's manual. In particular, when accessed from the C language, many calls return a characteristic −1 value when an error occurs, returning the error code in the global variable *errno.* Other languages may present errors in different ways.

A number of system standard types are defined in the include file <sys/types.h> and used in the specifications here and in many C programs. These include **caddr_t** giving a memory address (typically as a character pointer), **off_t** giving a file offset (typically as a long integer), and a set of unsigned types **u_char**, **u_short**, **u_int** and **u_long**, shorthand names for **unsigned char**, **unsigned short**, etc.

1. Kernel primitives

The facilities available to a UNIX user process are logically divided into two parts: kernel facilities directly implemented by UNIX code running in the operating system, and system facilities implemented either by the system, or in cooperation with a *server process.* These kernel facilities are described in this section 1.

The facilities implemented in the kernel are those which define the *UNIX virtual machine* which each process runs in. Like many real machines, this virtual machine has memory management hardware, an interrupt facility, timers and counters. The UNIX virtual machine also allows access to files and other objects through a set of *descriptors.* Each descriptor resembles a device controller, and supports a set of operations. Like devices on real machines, some of which are internal to the machine and some of which are external, parts of the descriptor machinery are built-in to the operating system, while other parts are often implemented in server processes on other machines. The facilities provided through the descriptor machinery are described in section 2.

1.1. Processes and protection

1.1.1. Host and process identifiers

Each UNIX host has associated with it a 32-bit host id, and a host name of up to 255 characters. These are set (by a privileged user) and returned by the calls:

```
sethostid(hostid)
long hostid;

hostid = gethostid();
result long hostid;

sethostname(name, len)
char *name; int len;

len = gethostname(buf, buflen)
result int len; result char *buf; int buflen;
```

On each host runs a set of *processes.* Each process is largely independent of other processes, having its own protection domain, address space, timers, and an independent set of references to system or user implemented objects.

Each process in a host is named by an integer called the *process id.* This number is in the range 1-30000 and is returned by the *getpid* routine:

```
pid = getpid();
result int pid;
```

On each UNIX host this identifier is guaranteed to be unique; in a multi-host environment, the (hostid, process id) pairs are guaranteed unique.

1.1.2. Process creation and termination

A new process is created by making a logical duplicate of an existing process:

```
pid = fork();
result int pid;
```

The *fork* call returns twice, once in the parent process, where *pid* is the process identifier of the child, and once in the child process where *pid* is 0. The parent-child relationship induces a hierarchical structure on the set of processes in the system.

A process may terminate by executing an *exit* call:

```
exit(status)
int status;
```

returning 8 bits of exit status to its parent.

When a child process exits or terminates abnormally, the parent process receives information about any event which caused termination of the child process. A second call provides a non-blocking interface and may also be used to retrieve information about resources consumed by the process during its lifetime.

```
#include <sys/wait.h>

pid = wait(astatus);
result int pid; result union wait *astatus;

pid = wait3(astatus, options, arusage);
result int pid; result union waitstatus *astatus;
int options; result struct rusage *arusage;
```

A process can overlay itself with the memory image of another process, passing the newly created process a set of parameters, using the call:

```
execve(name, argv, envp)
char *name, **argv, **envp;
```

The specified *name* must be a file which is in a format recognized by the system, either a binary executable file or a file which causes the execution of a specified interpreter program to process its contents.

1.1.3. User and group ids

Each process in the system has associated with it two user-id's: a *real user id* and a *effective user id*, both non-negative 16 bit integers. Each process has an *real accounting group id* and an *effective accounting group id* and a set of *access group id's*. The group id's are non-negative 16 bit integers. Each process may be in several different access groups, with the maximum concurrent number of access groups a system compilation parameter, the constant NGROUPS in the file <sys/param.h>, guaranteed to be at least 8.

The real and effective user ids associated with a process are returned by:

```
ruid = getuid();
result int ruid;

euid = geteuid();
result int euid;
```

the real and effective accounting group ids by:

```
rgid = getgid();
result int rgid;

egid = getegid();
result int egid;
```

and the access group id set is returned by a *getgroups* call:

```
ngroups = getgroups(gidsetsize, gidset);
result int ngroups; int gidsetsize; result int gidset[gidsetsize];
```

The user and group id's are assigned at login time using the *setreuid*, *setregid*, and *setgroups* calls:

```
setreuid(ruid, euid);
int ruid, euid;

setregid(rgid, egid);
int rgid, egid;

setgroups(gidsetsize, gidset)
int gidsetsize; int gidset[gidsetsize];
```

The *setreuid* call sets both the real and effective user-id's, while the *setregid* call sets both the real and effective accounting group id's. Unless the caller is the super-user, *ruid* must be equal to either the current real or effective user-id, and *rgid* equal to either the current real or effective accounting group id. The *setgroups* call is restricted to the super-user.

1.1.4. Process groups

Each process in the system is also normally associated with a *process group.* The group of processes in a process group is sometimes referred to as a *job* and manipulated by high-level system software (such as the shell). The current process group of a process is returned by the *getpgrp* call:

```
pgrp = getpgrp(pid);
result int pgrp; int pid;
```

When a process is in a specific process group it may receive software interrupts affecting the group, causing the group to suspend or resume execution or to be interrupted or terminated. In particular, a system terminal has a process group and only processes which are in the process group of the terminal may read from the terminal, allowing arbitration of terminals among several different jobs.

The process group associated with a process may be changed by the *setpgrp* call:

```
setpgrp(pid, pgrp);
int pid, pgrp;
```

Newly created processes are assigned process id's distinct from all processes and process groups, and the same process group as their parent. A normal (unprivileged) process may set its process group equal to its process id. A privileged process may set the process group of any process to any value.

1.2. Memory management†

1.2.1. Text, data and stack

Each process begins execution with three logical areas of memory called text, data and stack. The text area is read-only and shared, while the data and stack areas are private to the process. Both the data and stack areas may be extended and contracted on program request. The call

```
addr = sbrk(incr);
result caddr_t addr; int incr;
```

changes the size of the data area by *incr* bytes and returns the new end of the data area, while

```
addr = sstk(incr);
result caddr_t addr; int incr;
```

changes the size of the stack area. The stack area is also automatically extended as needed. On the VAX the text and data areas are adjacent in the P0 region, while the stack section is in the P1 region, and grows downward.

1.2.2. Mapping pages

The system supports sharing of data between processes by allowing pages to be mapped into memory. These mapped pages may be *shared* with other processes or *private* to the process. Protection and sharing options are defined in <mman.h> as:

```
/* protections are chosen from these bits, or-ed together */
#define  PROT_READ       0x4    /* pages can be read */
#define  PROT_WRITE      0x2    /* pages can be written */
#define  PROT_EXEC       0x1    /* pages can be executed */

/* sharing types; choose either SHARED or PRIVATE */
#define  MAP_SHARED      1      /* share changes */
#define  MAP_PRIVATE     2      /* changes are private */
```

The cpu-dependent size of a page is returned by the *getpagesize* system call:

```
pagesize = getpagesize();
result int pagesize;
```

The call:

```
mmap(addr, len, prot, share, fd, pos);
caddr_t addr; int len, prot, share, fd; off_t pos;
```

causes the pages starting at *addr* and continuing for *len* bytes to be mapped from the object represented by descriptor *fd*, at absolute position *pos*. The parameter *share* specifies whether modifications made to this mapped copy of the page, are to be kept *private*, or are to be *shared* with other references. The parameter *prot* specifies the accessibility of the mapped pages. The *addr*, *len*, and *pos* parameters must all be multiples of the pagesize.

A process can move pages within its own memory by using the *mremap* call:

```
mremap(addr, len, prot, share, fromaddr);
caddr_t addr; int len, prot, share; caddr_t fromaddr;
```

This call maps the pages starting at *fromaddr* to the address specified by *addr*.

† This section represents the interface planned for later releases of the system. Of the calls described in this section, only *sbrk* and *getpagesize* are included in 4.2BSD.

A mapping can be removed by the call

```
munmap(addr, len);
caddr_t addr; int len;
```

This causes further references to these pages to refer to private pages initialized to zero.

1.2.3. Page protection control

A process can control the protection of pages using the call

```
mprotect(addr, len, prot);
caddr_t addr; int len, prot;
```

This call changes the specified pages to have protection *prot*.

1.2.4. Giving and getting advice

A process that has knowledge of its memory behavior may use the *madvise* call:

```
madvise(addr, len, behav);
caddr_t addr; int len, behav;
```

Behav describes expected behavior, as given in <mman.h>:

```
#define MADV_NORMAL     0    /* no further special treatment */
#define MADV_RANDOM     1    /* expect random page references */
#define MADV_SEQUENTIAL 2    /* expect sequential references */
#define MADV_WILLNEED   3    /* will need these pages */
#define MADV_DONTNEED   4    /* don't need these pages */
```

Finally, a process may obtain information about whether pages are core resident by using the call

```
mincore(addr, len, vec)
caddr_t addr; int len; result char *vec;
```

Here the current core residency of the pages is returned in the character array *vec*, with a value of 1 meaning that the page is in-core.

1.3. Signals

1.3.1. Overview

The system defines a set of *signals* that may be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is blocked from further occurrence, the current process context is saved, and a new one is built. A process may specify the *handler* to which a signal is delivered, or specify that the signal is to be *blocked* or *ignored*. A process may also specify that a *default* action is to be taken when signals occur.

Some signals will cause a process to exit when they are not caught. This may be accompanied by creation of a *core* image file, containing the current memory image of the process for use in post-mortem debugging. A process may choose to have signals delivered on a special stack, so that sophisticated software stack manipulations are possible.

All signals have the same *priority*. If multiple signals are pending simultaneously, the order in which they are delivered to a process is implementation specific. Signal routines execute with the signal that caused their invocation *blocked*, but other signals may yet occur. Mechanisms are provided whereby critical sections of code may protect themselves against the occurrence of specified signals.

1.3.2. Signal types

The signals defined by the system fall into one of five classes: hardware conditions, software conditions, input/output notification, process control, or resource control. The set of signals is defined in the file <signal.h>.

Hardware signals are derived from exceptional conditions which may occur during execution. Such signals include SIGFPE representing floating point and other arithmetic exceptions, SIGILL for illegal instruction execution, SIGSEGV for addresses outside the currently assigned area of memory, and SIGBUS for accesses that violate memory protection constraints. Other, more cpu-specific hardware signals exist, such as those for the various customer-reserved instructions on the VAX (SIGIOT, SIGEMT, and SIGTRAP).

Software signals reflect interrupts generated by user request: SIGINT for the normal interrupt signal; SIGQUIT for the more powerful *quit* signal, that normally causes a core image to be generated; SIGHUP and SIGTERM that cause graceful process termination, either because a user has "hung up", or by user or program request; and SIGKILL, a more powerful termination signal which a process cannot catch or ignore. Other software signals (SIGALRM, SIGVTALRM, SIGPROF) indicate the expiration of interval timers.

A process can request notification via a SIGIO signal when input or output is possible on a descriptor, or when a *non-blocking* operation completes. A process may request to receive a SIGURG signal when an urgent condition arises.

A process may be *stopped* by a signal sent to it or the members of its process group. The SIGSTOP signal is a powerful stop signal, because it cannot be caught. Other stop signals SIGTSTP, SIGTTIN, and SIGTTOU are used when a user request, input request, or output request respectively is the reason the process is being stopped. A SIGCONT signal is sent to a process when it is continued from a stopped state. Processes may receive notification with a SIGCHLD signal when a child process changes state, either by stopping or by terminating.

Exceeding resource limits may cause signals to be generated. SIGXCPU occurs when a process nears its CPU time limit and SIGXFSZ warns that the limit on file size creation has been reached.

1.3.3. Signal handlers

A process has a handler associated with each signal that controls the way the signal is delivered. The call

```
#include <signal.h>

struct sigvec {
        int     (*sv_handler)();
        int     sv_mask;
        int     sv_onstack;
};

sigvec(signo, sv, osv)
int signo; struct sigvec *sv; result struct sigvec *osv;
```

assigns interrupt handler address *sv_handler* to signal *signo.* Each handler address specifies either an interrupt routine for the signal, that the signal is to be ignored, or that a default action (usually process termination) is to occur if the signal occurs. The constants SIG_IGN and SIG_DEF used as values for *sv_handler* cause ignoring or defaulting of a condition. The *sv_mask* and *sv_onstack* values specify the signal mask to be used when the handler is invoked and whether the handler should operate on the normal run-time stack or a special signal stack (see below). If *osv* is non-zero, the previous signal vector is returned.

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently *blocked* by the process then it will be delivered. The process of signal delivery adds the signal to be delivered and those signals specified in the associated signal handler's *sv_mask* to a set of those *masked* for the process, saves the current process context, and places the process in the context of the signal handling routine. The call is arranged so that if the signal handling routine exits normally the signal mask will be restored and the process will resume execution in the original context. If the process wishes to resume in a different context, then it must arrange to restore the signal mask itself.

The mask of *blocked* signals is independent of handlers for signals. It prevents signals from being delivered much as a raised hardware interrupt priority level prevents hardware interrupts. Preventing an interrupt from occurring by changing the handler is analogous to disabling a device from further interrupts.

The signal handling routine *sv_handler* is called by a C call of the form

```
(*sv_handler)(signo, code, scp);
int signo; long code; struct sigcontext *scp;
```

The *signo* gives the number of the signal that occurred, and the *code*, a word of information supplied by the hardware. The *scp* parameter is a pointer to a machine-dependent structure containing the information for restoring the context before the signal.

1.3.4. Sending signals

A process can send a signal to another process or group of processes with the calls:

```
kill(pid, signo)
int pid, signo;

killpgrp(pgrp, signo)
int pgrp, signo;
```

Unless the process sending the signal is privileged, it and the process receiving the signal must have the same effective user id.

Signals are also sent implicitly from a terminal device to the process group associated with the terminal when certain input characters are typed.

1.3.5. Protecting critical sections

To block a section of code against one or more signals, a *sigblock* call may be used to add a set of signals to the existing mask, returning the old mask:

```
oldmask = sigblock(mask);
result long oldmask; long mask;
```

The old mask can then be restored later with *sigsetmask*,

```
oldmask = sigsetmask(mask);
result long oldmask; long mask;
```

The *sigblock* call can be used to read the current mask by specifying an empty *mask*.

It is possible to check conditions with some signals blocked, and then to pause waiting for a signal and restoring the mask, by using:

```
sigpause(mask);
long mask;
```

1.3.6. Signal stacks

Applications that maintain complex or fixed size stacks can use the call

```
struct sigstack {
        caddr_t     ss_sp;
        int         ss_onstack;
};

sigstack(ss, oss)
struct sigstack *ss; result struct sigstack *oss;
```

to provide the system with a stack based at *ss_sp* for delivery of signals. The value *ss_onstack* indicates whether the process is currently on the signal stack, a notion maintained in software by the system.

When a signal is to be delivered, the system checks whether the process is on a signal stack. If not, then the process is switched to the signal stack for delivery, with the return from the signal arranged to restore the previous stack.

If the process wishes to take a non-local exit from the signal routine, or run code from the signal stack that uses a different stack, a *sigstack* call should be used to reset the signal stack.

1.4. Timers

1.4.1. Real time

The system's notion of the current Greenwich time and the current time zone is set and returned by the call by the calls:

```
#include <sys/time.h>

settimeofday(tvp, tzp);
struct timeval *tp;
struct timezone *tzp;

gettimeofday(tp, tzp);
result struct timeval *tp;
result struct timezone *tzp;
```

where the structures are defined in <sys/time.h> as:

```
struct timeval {
        long        tv_sec;        /* seconds since Jan 1, 1970 */
        long        tv_usec;       /* and microseconds */
};

struct timezone {
        int         tz_minuteswest;  /* of Greenwich */
        int         tz_dsttime;      /* type of dst correction to apply */
};
```

Earlier versions of UNIX contained only a 1-second resolution version of this call, which remains as a library routine:

```
time(tvsec)
result long *tvsec;
```

returning only the tv_sec field from the *gettimeofday* call.

1.4.2. Interval time

The system provides each process with three interval timers, defined in <sys/time.h>:

```
#define ITIMER_REAL      0    /* real time intervals */
#define ITIMER_VIRTUAL   1    /* virtual time intervals */
#define ITIMER_PROF      2    /* user and system virtual time */
```

The ITIMER_REAL timer decrements in real time. It could be used by a library routine to maintain a wakeup service queue. A SIGALRM signal is delivered when this timer expires.

The ITIMER_VIRTUAL timer decrements in process virtual time. It runs only when the process is executing. A SIGVTALRM signal is delivered when it expires.

The ITIMER_PROF timer decrements both in process virtual time and when the system is running on behalf of the process. It is designed to be used by processes to statistically profile their execution. A SIGPROF signal is delivered when it expires.

A timer value is defined by the *itimerval* structure:

```
struct itimerval {
        struct      timeval it_interval;  /* timer interval */
        struct      timeval it_value;     /* current value */
};
```

and a timer is set or read by the call:

```
getitimer(which, value);
int which; result struct itimerval *value;

setitimer(which, value, ovalue);
int which; struct itimerval *value; result struct itimerval *ovalue;
```

The third argument to *setitimer* specifies an optional structure to receive the previous contents of the interval timer. A timer can be disabled by specifying a timer value of 0.

The system rounds argument timer intervals to be not less than the resolution of its clock. This clock resolution can be determined by loading a very small value into a timer and reading the timer back to see what value resulted.

The *alarm* system call of earlier versions of UNIX is provided as a library routine using the ITIMER_REAL timer. The process profiling facilities of earlier versions of UNIX remain because it is not always possible to guarantee the automatic restart of system calls after receipt of a signal.

```
profil(buf, bufsize, offset, scale);
result char *buf; int bufsize, offset, scale;
```

1.5. Descriptors

1.5.1. The reference table

Each process has access to resources through *descriptors*. Each descriptor is a handle allowing the process to reference objects such as files, devices and communications links.

Rather than allowing processes direct access to descriptors, the system introduces a level of indirection, so that descriptors may be shared between processes. Each process has a *descriptor reference table*, containing pointers to the actual descriptors. The descriptors themselves thus have multiple references, and are reference counted by the system.

Each process has a fixed size descriptor reference table, where the size is returned by the *getdtablesize* call:

```
nds = getdtablesize();
result int nds;
```

and guaranteed to be at least 20. The entries in the descriptor reference table are referred to by small integers; for example if there are 20 slots they are numbered 0 to 19.

1.5.2. Descriptor properties

Each descriptor has a logical set of properties maintained by the system and defined by its *type*. Each type supports a set of operations; some operations, such as reading and writing, are common to several abstractions, while others are unique. The generic operations applying to many of these types are described in section 2.1. Naming contexts, files and directories are described in section 2.2. Section 2.3 describes communications domains and sockets. Terminals and (structured and unstructured) devices are described in section 2.4.

1.5.3. Managing descriptor references

A duplicate of a descriptor reference may be made by doing

```
new = dup(old);
result int new; int old;
```

returning a copy of descriptor reference *old* indistinguishable from the original. The *new* chosen by the system will be the smallest unused descriptor reference slot. A copy of a descriptor reference may be made in a specific slot by doing

```
dup2(old, new);
int old, new;
```

The *dup2* call causes the system to deallocate the descriptor reference current occupying slot *new*, if any, replacing it with a reference to the same descriptor as old. This deallocation is also performed by:

```
close(old);
int old;
```

1.5.4. Multiplexing requests

The system provides a standard way to do synchronous and asynchronous multiplexing of operations.

Synchronous multiplexing is performed by using the *select* call:

```
nds = select(nd, in, out, except, tvp);
result int nds; int nd; result *in, *out, *except;
struct timeval *tvp;
```

The *select* call examines the descriptors specified by the sets *in*, *out* and *except*, replacing the

specified bit masks by the subsets that select for input, output, and exceptional conditions respectively (*nd* indicates the size, in bytes, of the bit masks). If any descriptors meet the following criteria, then the number of such descriptors is returned in *nds* and the bit masks are updated.

- A descriptor selects for input if an input oriented operation such as *read* or *receive* is possible, or if a connection request may be accepted (see section 2.3.1.4).
- A descriptor selects for output if an output oriented operation such as *write* or *send* is possible, or if an operation that was "in progress", such as connection establishment, has completed (see section 2.1.3).
- A descriptor selects for an exceptional condition if a condition that would cause a SIGURG signal to be generated exists (see section 1.3.2).

If none of the specified conditions is true, the operation blocks for at most the amount of time specified by *tvp*, or waits for one of the conditions to arise if *tvp* is given as 0.

Options affecting i/o on a descriptor may be read and set by the call:

```
dopt = fcntl(d, cmd, arg)
result int dopt; int d, cmd, arg;

/* interesting values for cmd */
#define F_SETFL     3    /* set descriptor options */
#define F_GETFL     4    /* get descriptor options */
#define F_SETOWN    5    /* set descriptor owner (pid/pgrp) */
#define F_GETOWN    6    /* get descriptor owner (pid/pgrp) */
```

The F_SETFL *cmd* may be used to set a descriptor in non-blocking i/o mode and/or enable signalling when i/o is possible. F_SETOWN may be used to specify a process or process group to be signalled when using the latter mode of operation.

Operations on non-blocking descriptors will either complete immediately, note an error EWOULDBLOCK, partially complete an input or output operation returning a partial count, or return an error EINPROGRESS noting that the requested operation is in progress. A descriptor which has signalling enabled will cause the specified process and/or process group be signaled, with a SIGIO for input, output, or in-progress operation complete, or a SIGURG for exceptional conditions.

For example, when writing to a terminal using non-blocking output, the system will accept only as much data as there is buffer space for and return; when making a connection on a *socket*, the operation may return indicating that the connection establishment is "in progress". The *select* facility can be used to determine when further output is possible on the terminal, or when the connection establishment attempt is complete.

1.5.5. Descriptor wrapping.†

A user process may build descriptors of a specified type by *wrapping* a communications channel with a system supplied protocol translator:

```
new = wrap(old, proto)
result int new; int old; struct dprop *proto;
```

Operations on the descriptor *old* are then translated by the system provided protocol translator into requests on the underyling object *old* in a way defined by the protocol. The protocols supported by the kernel may vary from system to system and are described in the programmers manual.

† The facilities described in this section are not included in 4.2BSD.

Protocols may be based on communications multiplexing or a rights-passing style of handling multiple requests made on the same object. For instance, a protocol for implementing a file abstraction may or may not include locally generated "read-ahead" requests. A protocol that provides for read-ahead may provide higher performance but have a more difficult implementation.

Another example is the terminal driving facilities. Normally a terminal is associated with a communications line and the terminal type and standard terminal access protocol is wrapped around a synchronous communications line and given to the user. If a virtual terminal is required, the terminal driver can be wrapped around a communications link, the other end of which is held by a virtual terminal protocol interpreter.

1.6. Resource controls

1.6.1. Process priorities

The system gives CPU scheduling priority to processes that have not used CPU time recently. This tends to favor interactive processes and processes that execute only for short periods. It is possible to determine the priority currently assigned to a process, process group, or the processes of a specified user, or to alter this priority using the calls:

```
#define PRIO_PROCESS     0     /* process */
#define PRIO_PGRP        1     /* process group */
#define PRIO_USER        2     /* user id */

prio = getpriority(which, who);
result int prio; int which, who;

setpriority(which, who, prio);
int which, who, prio;
```

The value *prio* is in the range −20 to 20. The default priority is 0; lower priorities cause more favorable execution. The *getpriority* call returns the highest priority (lowest numerical value) enjoyed by any of the specified processes. The *setpriority* call sets the priorities of all of the specified processes to the specified value. Only the super-user may lower priorities.

1.6.2. Resource utilization

The resources used by a process are returned by a *getrusage* call, returning information in a structure defined in <sys/resource.h>:

```
#define RUSAGE_SELF      0     /* usage by this process */
#define RUSAGE_CHILDREN -1     /* usage by all children */

getrusage(who, rusage)
int who; result struct rusage *rusage;

struct rusage {
        struct  timeval ru_utime;  /* user time used */
        struct  timeval ru_stime;  /* system time used */
        int     ru_maxrss;         /* maximum core resident set size: kbytes */
        int     ru_ixrss;          /* integral shared memory size (kbytes*sec) */
        int     ru_idrss;          /* unshared data " */
        int     ru_isrss;          /* unshared stack " */
        int     ru_minflt;         /* page-reclaims */
        int     ru_majflt;         /* page faults */
        int     ru_nswap;          /* swaps */
        int     ru_inblock;        /* block input operations */
        int     ru_oublock;        /* block output " */
        int     ru_msgsnd;         /* messages sent */
        int     ru_msgrcv;         /* messages received */
        int     ru_nsignals;       /* signals received */
        int     ru_nvcsw;          /* voluntary context switches */
        int     ru_nivcsw;         /* involuntary " */
};
```

The *who* parameter specifies whose resource usage is to be returned. The resources used by the current process, or by all the terminated children of the current process may be requested.

1.6.3. Resource limits

The resources of a process for which limits are controlled by the kernel are defined in <sys/resource.h>, and controlled by the *getrlimit* and *setrlimit* calls:

```
#define RLIMIT_CPU       0      /* cpu time in milliseconds */
#define RLIMIT_FSIZE     1      /* maximum file size */
#define RLIMIT_DATA      2      /* maximum data segment size */
#define RLIMIT_STACK     3      /* maximum stack segment size */
#define RLIMIT_CORE      4      /* maximum core file size */
#define RLIMIT_RSS       5      /* maximum resident set size */

#define RLIM_NLIMITS     6

#define RLIM_INFINITY    0x7fffffff

struct rlimit {
        int     rlim_cur;       /* current (soft) limit */
        int     rlim_max;       /* hard limit */
};

getrlimit(resource, rlp)
int resource; result struct rlimit *rlp;

setrlimit(resource, rlp)
int resource; struct rlimit *rlp;
```

Only the super-user can raise the maximum limits. Other users may only alter *rlim_cur* within the range from 0 to *rlim_max* or (irreversibly) lower *rlim_max*.

1.7. System operation support

Unless noted otherwise, the calls in this section are permitted only to a privileged user.

1.7.1. Bootstrap operations

The call

```
mount(blkdev, dir, ronly);
char *blkdev, *dir; int ronly;
```

extends the UNIX name space. The *mount* call specifies a block device *blkdev* containing a UNIX file system to be made available starting at *dir*. If *ronly* is set then the file system is read-only; writes to the file system will not be permitted and access times will not be updated when files are referenced. *Dir* is normally a name in the root directory.

The call

```
swapon(blkdev, size);
char *blkdev; int size;
```

specifies a device to be made available for paging and swapping.

1.7.2. Shutdown operations

The call

```
unmount(dir);
char *dir;
```

unmounts the file system mounted on *dir*. This call will succeed only if the file system is not currently being used.

The call

```
sync();
```

schedules input/output to clean all system buffer caches. (This call does not require priveleged status.)

The call

```
reboot(how)
int how;
```

causes a machine halt or reboot. The call may request a reboot by specifying *how* as RB_AUTOBOOT, or that the machine be halted with RB_HALT. These constants are defined in <sys/reboot.h>.

1.7.3. Accounting

The system optionally keeps an accounting record in a file for each process that exits on the system. The format of this record is beyond the scope of this document. The accounting may be enabled to a file *name* by doing

```
acct(path);
char *path;
```

If *path* is null, then accounting is disabled. Otherwise, the named file becomes the accounting file.

2. System facilities

This section discusses the system facilities that are not considered part of the kernel.

The system abstractions described are:

Directory contexts
: A directory context is a position in the UNIX file system name space. Operations on files and other named objects in a file system are always specified relative to such a context.

Files
: Files are used to store uninterpreted sequence of bytes on which random access *reads* and *writes* may occur. Pages from files may also be mapped into process address space. A directory may be read as a file†.

Communications domains
: A communications domain represents an interprocess communications environment, such as the communications facilities of the UNIX system, communications in the INTERNET, or the resource sharing protocols and access rights of a resource sharing system on a local network.

Sockets
: A socket is an endpoint of communication and the focal point for IPC in a communications domain. Sockets may be created in pairs, or given names and used to rendezvous with other sockets in a communications domain, accepting connections from these sockets or exchanging messages with them. These operations model a labeled or unlabeled communications graph, and can be used in a wide variety of communications domains. Sockets can have different *types* to provide different semantics of communication, increasing the flexibility of the model.

Terminals and other devices
: Devices include terminals, providing input editing and interrupt generation and output flow control and editing, magnetic tapes, disks and other peripherals. They often support the generic *read* and *write* operations as well as a number of *ioctl*s.

Processes
: Process descriptors provide facilities for control and debugging of other processes.

† Support for mapping files is not included in the 4.2 release.

2.1. Generic operations

Many system abstractions support the operations *read*, *write* and *ioctl*. We describe the basics of these common primitives here. Similarly, the mechanisms whereby normally synchronous operations may occur in a non-blocking or asynchronous fashion are common to all system-defined abstractions and are described here.

2.1.1. Read and write

The *read* and *write* system calls can be applied to communications channels, files, terminals and devices. They have the form:

```
cc = read(fd, buf, nbytes);
result int cc; int fd; result caddr_t buf; int nbytes;

cc = write(fd, buf, nbytes);
result int cc; int fd; caddr_t buf; int nbytes;
```

The *read* call transfers as much data as possible from the object defined by *fd* to the buffer at address *buf* of size *nbytes*. The number of bytes transferred is returned in *cc*, which is −1 if a return occurred before any data was transferred because of an error or use of non-blocking operations.

The *write* call transfers data from the buffer to the object defined by *fd*. Depending on the type of *fd*, it is possible that the *write* call will accept some portion of the provided bytes; the user should resubmit the other bytes in a later request in this case. Error returns because of interrupted or otherwise incomplete operations are possible.

Scattering of data on input or gathering of data for output is also possible using an array of input/output vector descriptors. The type for the descriptors is defined in <sys/uio.h> as:

```
struct iovec {
        caddr_t    iov_msg;        /* base of a component */
        int        iov_len;        /* length of a component */
};
```

The calls using an array of descriptors are:

```
cc = readv(fd, iov, iovlen);
result int cc; int fd; struct iovec *iov; int iovlen;

cc = writev(fd, iov, iovlen);
result int cc; int fd; struct iovec *iov; int iovlen;
```

Here *iovlen* is the count of elements in the *iov* array.

2.1.2. Input/output control

Control operations on an object are performed by the *ioctl* operation:

```
ioctl(fd, request, buffer);
int fd, request; caddr_t buffer;
```

This operation causes the specified *request* to be performed on the object *fd*. The *request* parameter specifies whether the argument buffer is to be read, written, read and written, or is not needed, and also the size of the buffer, as well as the request. Different descriptor types and subtypes within descriptor types may use distinct *ioctl* requests. For example, operations on terminals control flushing of input and output queues and setting of terminal parameters; operations on disks cause formatting operations to occur; operations on tapes control tape positioning.

The names for basic control operations are defined in <sys/ioctl.h>.

2.1.3. Non-blocking and asynchronous operations

A process that wishes to do non-blocking operations on one of its descriptors sets the descriptor in non-blocking mode as described in section 1.5.4. Thereafter the *read* call will return a specific EWOULDBLOCK error indication if there is no data to be *read.* The process may *dselect* the associated descriptor to determine when a read is possible.

Output attempted when a descriptor can accept less than is requested will either accept some of the provided data, returning a shorter than normal length, or return an error indicating that the operation would block. More output can be performed as soon as a *select* call indicates the object is writeable.

Operations other than data input or output may be performed on a descriptor in a non-blocking fashion. These operations will return with a characteristic error indicating that they are in progress if they cannot return immediately. The descriptor may then be *select*ed for *write* to find out when the operation can be retried. When *select* indicates the descriptor is writeable, a respecification of the original operation will return the result of the operation.

2.2. File system

2.2.1. Overview

The file system abstraction provides access to a hierarchical file system structure. The file system contains directories (each of which may contain other sub-directories) as well as files and references to other objects such as devices and inter-process communications sockets.

Each file is organized as a linear array of bytes. No record boundaries or system related information is present in a file. Files may be read and written in a random-access fashion. The user may read the data in a directory as though it were an ordinary file to determine the names of the contained files, but only the system may write into the directories. The file system stores only a small amount of ownership, protection and usage information with a file.

2.2.2. Naming

The file system calls take *path name* arguments. These consist of a zero or more component *file names* separated by "/" characters, where each file name is up to 255 ASCII characters excluding null and "/".

Each process always has two naming contexts: one for the root directory of the file system and one for the current working directory. These are used by the system in the filename translation process. If a path name begins with a "/", it is called a full path name and interpreted relative to the root directory context. If the path name does not begin with a "/" it is called a relative path name and interpreted relative to the current directory context.

The system limits the total length of a path name to 1024 characters.

The file name ".." in each directory refers to the parent directory of that directory. The parent directory of a file system is always the systems root directory.

The calls

```
chdir(path);
char *path;

chroot(path)
char *path;
```

change the current working directory and root directory context of a process. Only the super-user can change the root directory context of a process.

2.2.3. Creation and removal

The file system allows directories, files, special devices, and "portals" to be created and removed from the file system.

2.2.3.1. Directory creation and removal

A directory is created with the *mkdir* system call:

```
mkdir(path, mode);
char *path; int mode;
```

and removed with the *rmdir* system call:

```
rmdir(path);
char *path;
```

A directory must be empty if it is to be deleted.

2.2.3.2. File creation

Files are created with the *open* system call,

```
fd = open(path, oflag, mode);
result int fd; char *path; int oflag, mode;
```

The *path* parameter specifies the name of the file to be created. The *oflag* parameter must include O_CREAT from below to cause the file to be created. The protection for the new file is specified in *mode*. Bits for *oflag* are defined in <sys/file.h>:

```
#define O_RDONLY     000     /* open for reading */
#define O_WRONLY     001     /* open for writing */
#define O_RDWR       002     /* open for read & write */
#define O_NDELAY     004     /* non-blocking open */
#define O_APPEND     010     /* append on each write */
#define O_CREAT      01000   /* open with file create */
#define O_TRUNC      02000   /* open with truncation */
#define O_EXCL       04000   /* error on create if file exists */
```

One of O_RDONLY, O_WRONLY and O_RDWR should be specified, indicating what types of operations are desired to be performed on the open file. The operations will be checked against the user's access rights to the file before allowing the *open* to succeed. Specifying O_APPEND causes writes to automatically append to the file. The flag O_CREAT causes the file to be created if it does not exist, with the specified *mode*, owned by the current user and the group of the containing directory.

If the open specifies to create the file with O_EXCL and the file already exists, then the *open* will fail without affecting the file in any way. This provides a simple exclusive access facility.

2.2.3.3. Creating references to devices

The file system allows entries which reference peripheral devices. Peripherals are distinguished as *block* or *character* devices according by their ability to support block-oriented operations. Devices are identified by their "major" and "minor" device numbers. The major device number determines the kind of peripheral it is, while the minor device number indicates one of possibly many peripherals of that kind. Structured devices have all operations performed internally in "block" quantities while unstructured devices often have a number of special *ioctl* operations, and may have input and output performed in large units. The *mknod* call creates special entries:

```
mknod(path, mode, dev);
char *path; int mode, dev;
```

where *mode* is formed from the object type and access permissions. The parameter *dev* is a configuration dependent parameter used to identify specific character or block i/o devices.

2.2.3.4. Portal creation†

The call

```
fd = portal(name, server, param, dtype, protocol, domain, socktype)
result int fd; char *name, *server, *param; int dtype, protocol;
int domain, socktype;
```

places a *name* in the file system name space that causes connection to a server process when the name is used. The portal call returns an active portal in *fd* as though an access had occurred to

† The *portal* call is not implemented in 4.2BSD.

activate an inactive portal, as now described.

When an inactive portal is accesseed, the system sets up a socket of the specified *socktype* in the specified communications *domain* (see section 2.3), and creates the *server* process, giving it the specified *param* as argument to help it identify the portal, and also giving it the newly created socket as descriptor number 0. The accessor of the portal will create a socket in the same *domain* and *connect* to the server. The user will then *wrap* the socket in the specified *protocol* to create an object of the required descriptor type *dtype* and proceed with the operation which was in progress before the portal was encountered.

While the server process holds the socket (which it received as *fd* from the *portal* call on descriptor 0 at activation) further references will result in connections being made to the same socket.

2.2.3.5. File, device, and portal removal

A reference to a file, special device or portal may be removed with the *unlink* call,

```
unlink(path);
char *path;
```

The caller must have write access to the directory in which the file is located for this call to be successful.

2.2.4. Reading and modifying file attributes

Detailed information about the attributes of a file may be obtained with the calls:

```
#include <sys/stat.h>

stat(path, stb);
char *path; result struct stat *stb;

fstat(fd, stb);
int fd; result struct stat *stb;
```

The *stat* structure includes the file type, protection, ownership, access times, size, and a count of hard links. If the file is a symbolic link, then the status of the link itself (rather than the file the link references) may be found using the *lstat* call:

```
lstat(path, stb);
char *path; result struct stat *stb;
```

Newly created files are assigned the user id of the process that created it and the group id of the directory in which it was created. The ownership of a file may be changed by either of the calls

```
chown(path, owner, group);
char *path; int owner, group;

fchown(fd, owner, group);
int fd, owner, group;
```

In addition to ownership, each file has three levels of access protection associated with it. These levels are owner relative, group relative, and global (all users and groups). Each level of access has separate indicators for read permission, write permission, and execute permission. The protection bits associated with a file may be set by either of the calls:

```
chmod(path, mode);
char *path; int mode;

fchmod(fd, mode);
int fd, mode;
```

where *mode* is a value indicating the new protection of the file. The file mode is a three digit octal number. Each digit encodes read access as 4, write access as 2 and execute access as 1, or'ed together. The 0700 bits describe owner access, the 070 bits describe the access rights for processes in the same group as the file, and the 07 bits describe the access rights for other processes.

Finally, the access and modify times on a file may be set by the call:

```
utimes(path, tvp)
char *path; struct timeval *tvp[2];
```

This is particularly useful when moving files between media, to preserve relationships between the times the file was modified.

2.2.5. Links and renaming

Links allow multiple names for a file to exist. Links exist independently of the file linked to.

Two types of links exist, *hard* links and *symbolic* links. A hard link is a reference counting mechanism that allows a file to have multiple names within the same file system. Symbolic links cause string substitution during the pathname interpretation process.

Hard links and symbolic links have different properties. A hard link insures the target file will always be accessible, even after its original directory entry is removed; no such guarantee exists for a symbolic link. Symbolic links can span file systems boundaries.

The following calls create a new link, named *path2*, to *path1*:

```
link(path1, path2);
char *path1, *path2;

symlink(path1, path2);
char *path1, *path2;
```

The *unlink* primitive may be used to remove either type of link.

If a file is a symbolic link, the "value" of the link may be read with the *readlink* call,

```
len = readlink(path, buf, bufsize);
result int len; result char *path, *buf; int bufsize;
```

This call returns, in *buf*, the null-terminated string substituted into pathnames passing through *path*.

Atomic renaming of file system resident objects is possible with the *rename* call:

```
rename(oldname, newname);
char *oldname, *newname;
```

where both *oldname* and *newname* must be in the same file system. If *newname* exists and is a directory, then it must be empty.

2.2.6. Extension and truncation

Files are created with zero length and may be extended simply by writing or appending to them. While a file is open the system maintains a pointer into the file indicating the current location in the file associated with the descriptor. This pointer may be moved about in the file

in a random access fashion. To set the current offset into a file, the *lseek* call may be used,

```
oldoffset = lseek(fd, offset, type);
result off_t oldoffset; int fd; off_t offset; int type;
```

where *type* is given in <sys/file.h> as one of,

```
#define L_SET    0    /* set absolute file offset */
#define L_INCR   1    /* set file offset relative to current position */
#define L_XTND   2    /* set offset relative to end-of-file */
```

The call "lseek(fd, 0, L_INCR)" returns the current offset into the file.

Files may have "holes" in them. Holes are void areas in the linear extent of the file where data has never been written. These may be created by seeking to a location in a file past the current end-of-file and writing. Holes are treated by the system as zero valued bytes.

A file may be truncated with either of the calls:

```
truncate(path, length);
char *path; int length;

ftruncate(fd, length);
int fd, length;
```

reducing the size of the specified file to *length* bytes.

2.2.7. Checking accessibility

A process running with different real and effective user ids may interrogate the accessibility of a file to the real user by using the *access* call:

```
accessible = access(path, how);
result int accessible; char *path; int how;
```

Here *how* is constructed by or'ing the following bits, defined in <sys/file.h>:

```
#define F_OK    0    /* file exists */
#define X_OK    1    /* file is executable */
#define W_OK    2    /* file is writable */
#define R_OK    4    /* file is readable */
```

The presence or absence of advisory locks does not affect the result of *access*.

2.2.8. Locking

The file system provides basic facilities that allow cooperating processes to synchronize their access to shared files. A process may place an advisory *read* or *write* lock on a file, so that other cooperating processes may avoid interfering with the process' access. This simple mechanism provides locking with file granularity. More granular locking can be built using the IPC facilities to provide a lock manager. The system does not force processes to obey the locks; they are of an advisory nature only.

Locking is performed after an *open* call by applying the *flock* primitive,

```
flock(fd, how);
int fd, how;
```

where the *how* parameter is formed from bits defined in <sys/file.h>:

```
#define LOCK_SH    1    /* shared lock */
#define LOCK_EX    2    /* exclusive lock */
#define LOCK_NB    4    /* don't block when locking */
#define LOCK_UN    8    /* unlock */
```

Successive lock calls may be used to increase or decrease the level of locking. If an object is currently locked by another process when a *flock* call is made, the caller will be blocked until the current lock owner releases the lock; this may be avoided by including LOCK_NB in the *how* parameter. Specifying LOCK_UN removes all locks associated with the descriptor. Advisory locks held by a process are automatically deleted when the process terminates.

2.2.9. Disk quotas

As an optional facility, each file system may be requested to impose limits on a user's disk usage. Two quantities are limited: the total amount of disk space which a user may allocate in a file system and the total number of files a user may create in a file system. Quotas are expressed as *hard* limits and *soft* limits. A hard limit is always imposed; if a user would exceed a hard limit, the operation which caused the resource request will fail. A soft limit results in the user receiving a warning message, but with allocation succeeding. Facilities are provided to turn soft limits into hard limits if a user has exceeded a soft limit for an unreasonable period of time.

To enable disk quotas on a file system the *setquota* call is used:

```
setquota(special, file)
char *special, *file;
```

where *special* refers to a structured device file where a mounted file system exists, and *file* refers to a disk quota file (residing on the file system associated with *special*) from which user quotas should be obtained. The format of the disk quota file is implementation dependent.

To manipulate disk quotas the *quota* call is provided:

```
#include <sys/quota.h>

quota(cmd, uid, arg, addr)
int cmd, uid, arg; caddr_t addr;
```

The indicated *cmd* is applied to the user ID *uid*. The parameters *arg* and *addr* are command specific. The file <sys/quota.h> contains definitions pertinent to the use of this call.

2.3. Interprocess communications

2.3.1. Interprocess communication primitives

2.3.1.1. Communication domains

The system provides access to an extensible set of communication *domains*. A communication domain is identified by a manifest constant defined in the file <sys/socket.h>. Important standard domains supported by the system are the "unix" domain, AF_UNIX, for communication within the system, and the "internet" domain for communication in the DARPA internet, AF_INET. Other domains can be added to the system.

2.3.1.2. Socket types and protocols

Within a domain, communication takes place between communication endpoints known as *sockets*. Each socket has the potential to exchange information with other sockets within the domain.

Each socket has an associated abstract type, which describes the semantics of communication using that socket. Properties such as reliability, ordering, and prevention of duplication of messages are determined by the type. The basic set of socket types is defined in <sys/socket.h>:

```
/* Standard socket types */
#define SOCK_DGRAM      1    /* datagram */
#define SOCK_STREAM     2    /* virtual circuit */
#define SOCK_RAW        3    /* raw socket */
#define SOCK_RDM        4    /* reliably-delivered message */
#define SOCK_SEQPACKET  5    /* sequenced packets */
```

The SOCK_DGRAM type models the semantics of datagrams in network communication: messages may be lost or duplicated and may arrive out-of-order. The SOCK_RDM type models the semantics of reliable datagrams: messages arrive unduplicated and in-order, the sender is notified if messages are lost. The *send* and *receive* operations (described below) generate reliable/unreliable datagrams. The SOCK_STREAM type models connection-based virtual circuits: two-way byte streams with no record boundaries. The SOCK_SEQPACKET type models a connection-based, full-duplex, reliable, sequenced packet exchange; the sender is notified if messages are lost, and messages are never duplicated or presented out-of-order. Users of the last two abstractions may use the facilities for out-of-band transmission to send out-of-band data.

SOCK_RAW is used for unprocessed access to internal network layers and interfaces; it has no specific semantics.

Other socket types can be defined.†

Each socket may have a concrete *protocol* associated with it. This protocol is used within the domain to provide the semantics required by the socket type. For example, within the "internet" domain, the SOCK_DGRAM type may be implemented by the UDP user datagram protocol, and the SOCK_STREAM type may be implemented by the TCP transmission control protocol, while no standard protocols to provide SOCK_RDM or SOCK_SEQPACKET sockets exist.

† 4.2BSD does not support the SOCK_RDM and SOCK_SEQPACKET types.

2.3.1.3. Socket creation, naming and service establishment

Sockets may be *connected* or *unconnected*. An unconnected socket descriptor is obtained by the *socket* call:

```
s = socket(domain, type, protocol);
result int s; int domain, type, protocol;
```

An unconnected socket descriptor may yield a connected socket descriptor in one of two ways: either by actively connecting to another socket, or by becoming associated with a name in the communications domain and *accepting* a connection from another socket.

To accept connections, a socket must first have a binding to a name within the communications domain. Such a binding is established by a *bind* call:

```
bind(s, name, namelen);
int s; char *name; int namelen;
```

A socket's bound name may be retrieved with a *getsockname* call:

```
getsockname(s, name, namelen);
int s; result caddr_t name; result int *namelen;
```

while the peer's name can be retrieved with *getpeername*:

```
getpeername(s, name, namelen);
int s; result caddr_t name; result int *namelen;
```

Domains may support sockets with several names.

2.3.1.4. Accepting connections

Once a binding is made, it is possible to *listen* for connections:

```
listen(s, backlog);
int s, backlog;
```

The *backlog* specifies the maximum count of connections that can be simultaneously queued awaiting acceptance.

An *accept* call:

```
t = accept(s, name, anamelen);
result int t; int s; result caddr_t name; result int *anamelen;
```

returns a descriptor for a new, connected, socket from the queue of pending connections on *s*.

2.3.1.5. Making connections

An active connection to a named socket is made by the *connect* call:

```
connect(s, name, namelen);
int s; caddr_t name; int namelen;
```

It is also possible to create connected pairs of sockets without using the domain's name space to rendezvous; this is done with the *socketpair* call†:

```
socketpair(d, type, protocol, sv);
int d, type, protocol; result int sv[2];
```

Here the returned *sv* descriptors correspond to those obtained with *accept* and *connect*.

---

† 4.2BSD supports *socketpair* creation only in the "unix" communication domain.

The call

```
pipe(pv)
result int pv[2];
```

creates a pair of SOCK_STREAM sockets in the UNIX domain, with pv[0] only writeable and pv[1] only readable.

2.3.1.6. Sending and receiving data

Messages may be sent from a socket by:

```
cc = sendto(s, buf, len, flags, to, tolen);
result int cc; int s; caddr_t buf; int len, flags; caddr_t to; int tolen;
```

if the socket is not connected or:

```
cc = send(s, buf, len, flags);
result int cc; int s; caddr_t buf; int len, flags;
```

if the socket is connected. The corresponding receive primitives are:

```
msglen = recvfrom(s, buf, len, flags, from, fromlenaddr);
result int msglen; int s; result caddr_t buf; int len, flags;
result caddr_t from; result int *fromlenaddr;
```

and

```
msglen = recv(s, buf, len, flags);
result int msglen; int s; result caddr_t buf; int len, flags;
```

In the unconnected case, the parameters *to* and *tolen* specify the destination or source of the message, while the *from* parameter stores the source of the message, and **fromlenaddr* initially gives the size of the *from* buffer and is updated to reflect the true length of the *from* address.

All calls cause the message to be received in or sent from the message buffer of length *len* bytes, starting at address *buf*. The *flags* specify peeking at a message without reading it or sending or receiving high-priority out-of-band messages, as follows:

```
#define  MSG_PEEK     0x1   /* peek at incoming message */
#define  MSG_OOB      0x2   /* process out-of-band data */
```

2.3.1.7. Scatter/gather and exchanging access rights

It is possible scatter and gather data and to exchange access rights with messages. When either of these operations is involved, the number of parameters to the call becomes large. Thus the system defines a message header structure, in <sys/socket.h>, which can be used to conveniently contain the parameters to the calls:

```
struct msghdr {
        caddr_t   msg_name;            /* optional address */
        int       msg_namelen;         /* size of address */
        struct    iov *msg_iov;        /* scatter/gather array */
        int       msg_iovlen;          /* # elements in msg_iov */
        caddr_t   msg_accrights;       /* access rights sent/received */
        int       msg_accrightslen;    /* size of msg_accrights */
};
```

Here *msg_name* and *msg_namelen* specify the source or destination address if the socket is unconnected; *msg_name* may be given as a null pointer if no names are desired or required. The *msg_iov* and *msg_iovlen* describe the scatter/gather locations, as described in section 2.1.3.

Access rights to be sent along with the message are specified in *msg_accrights*, which has length *msg_accrightslen*. In the "unix" domain these are an array of integer descriptors, taken from the sending process and duplicated in the receiver.

This structure is used in the operations *sendmsg* and *recvmsg*:

```
sendmsg(s, msg, flags);
int s; struct msghdr *msg; int flags;

msglen = recvmsg(s, msg, flags);
result int msglen; int s; result struct msghdr *msg; int flags;
```

2.3.1.8. Using read and write with sockets

The normal UNIX *read* and *write* calls may be applied to connected sockets and translated into *send* and *receive* calls from or to a single area of memory and discarding any rights received. A process may operate on a virtual circuit socket, a terminal or a file with blocking or non-blocking input/output operations without distinguishing the descriptor type.

2.3.1.9. Shutting down halves of full-duplex connections

A process that has a full-duplex socket such as a virtual circuit and no longer wishes to read from or write to this socket can give the call:

```
shutdown(s, direction);
int s, direction;
```

where *direction* is 0 to not read further, 1 to not write further, or 2 to completely shut the connection down.

2.3.1.10. Socket and protocol options

Sockets, and their underlying communication protocols, may support *options*. These options may be used to manipulate implementation specific or non-standard facilities. The *getsockopt* and *setsockopt* calls are used to control options:

```
getsockopt(s, level, optname, optval, optlen)
int s, level, optname; result caddr_t optval; result int *optlen;

setsockopt(s, level, optname, optval, optlen)
int s, level, optname; caddr_t optval; int optlen;
```

The option *optname* is interpreted at the indicated protocol *level* for socket *s*. If a value is specified with *optval* and *optlen*, it is interpreted by the software operating at the specified *level*. The *level* SOL_SOCKET is reserved to indicate options maintained by the socket facilities. Other *level* values indicate a particular protocol which is to act on the option request; these values are normally interpreted as a "protocol number".

2.3.2. UNIX domain

This section describes briefly the properties of the UNIX communications domain.

2.3.2.1. Types of sockets

In the UNIX domain, the SOCK_STREAM abstraction provides pipe-like facilities, while SOCK_DGRAM provides (usually) reliable message-style communications.

2.3.2.2. Naming

Socket names are strings and may appear in the UNIX file system name space through portals†.

† The 4.2BSD implementation of the UNIX domain embeds bound sockets in the UNIX file system name space; this is a side effect of the implementation.

2.3.2.3. Access rights transmission

The ability to pass UNIX descriptors with messages in this domain allows migration of service within the system and allows user processes to be used in building system facilities.

2.3.3. INTERNET domain

This section describes briefly how the INTERNET domain is mapped to the model described in this section. More information will be found in the document describing the network implementation in 4.2BSD.

2.3.3.1. Socket types and protocols

SOCK_STREAM is supported by the INTERNET TCP protocol; SOCK_DGRAM by the UDP protocol. The SOCK_SEQPACKET has no direct INTERNET family analogue; a protocol based on one from the XEROX NS family and layered on top of IP could be implemented to fill this gap.

2.3.3.2. Socket naming

Sockets in the INTERNET domain have names composed of the 32 bit internet address, and a 16 bit port number. Options may be used to provide source routing for the address, security options, or additional address for subnets of INTERNET for which the basic 32 bit addresses are insufficient.

2.3.3.3. Access rights transmission

No access rights transmission facilities are provided in the INTERNET domain.

2.3.3.4. Raw access

The INTERNET domain allows the super-user access to the raw facilities of the various network interfaces and the various internal layers of the protocol implementation. This allows administrative and debugging functions to occur. These interfaces are modeled as SOCK_RAW sockets.

2.4. Terminals and Devices

2.4.1. Terminals

Terminals support *read* and *write* i/o operations, as well as a collection of terminal specific *ioctl* operations, to control input character editing, and output delays.

2.4.1.1. Terminal input

Terminals are handled according to the underlying communication characteristics such as baud rate and required delays, and a set of software parameters.

2.4.1.1.1. Input modes

A terminal is in one of three possible modes: *raw*, *cbreak*, or *cooked*. In raw mode all input is passed through to the reading process immediately and without interpretation. In cbreak mode, the handler interprets input only by looking for characters that cause interrupts or output flow control; all other characters are made available as in raw mode. In cooked mode, input is processed to provide standard line-oriented local editing functions, and input is presented on a line-by-line basis.

2.4.1.1.2. Interrupt characters

Interrupt characters are interpreted by the terminal handler only in cbreak and cooked modes, and cause a software interrupt to be sent to all processes in the process group associated with the terminal. Interrupt characters exist to send SIGINT and SIGQUIT signals, and to stop a process group with the SIGTSTP signal either immediately, or when all input up to the stop character has been read.

2.4.1.1.3. Line editing

When the terminal is in cooked mode, editing of an input line is performed. Editing facilities allow deletion of the previous character or word, or deletion of the current input line. In addition, a special character may be used to reprint the current input line after some number of editing operations have been applied.

Certain other characters are interpreted specially when a process is in cooked mode. The *end of line* character determines the end of an input record. The *end of file* character simulates an end of file occurrence on terminal input. Flow control is provided by *stop output* and *start output* control characters. Output may be flushed with the *flush output* character; and a *literal character* may be used to force literal input of the immediately following character in the input line.

2.4.1.2. Terminal output

On output, the terminal handler provides some simple formatting services. These include converting the carriage return character to the two character return-linefeed sequence, displaying non-graphic ASCII characters as "^character", inserting delays after certain standard control characters, expanding tabs, and providing translations for upper-case only terminals.

2.4.1.3. Terminal control operations

When a terminal is first opened it is initialized to a standard state and configured with a set of standard control, editing, and interrupt characters. A process may alter this configuration with certain control operations, specifying parameters in a standard structure:

```
struct ttymode {
        short     tt_ispeed;      /* input speed */
        int       tt_iflags;      /* input flags */
        short     tt_ospeed;      /* output speed */
        int       tt_oflags;      /* output flags */
};
```

and "special characters" are specified with the *ttychars* structure,

```
struct ttychars {
        char      tc_erasec;      /* erase char */
        char      tc_killc;       /* erase line */
        char      tc_intrc;       /* interrupt */
        char      tc_quitc;       /* quit */
        char      tc_startc;      /* start output */
        char      tc_stopc;       /* stop output */
        char      tc_eofc;        /* end-of-file */
        char      tc_brkc;        /* input delimiter (like nl) */
        char      tc_suspc;       /* stop process signal */
        char      tc_dsuspc;      /* delayed stop process signal */
        char      tc_rprntc;      /* reprint line */
        char      tc_flushc;      /* flush output (toggles) */
        char      tc_werasc;      /* word erase */
        char      tc_lnextc;      /* literal next character */
};
```

2.4.1.4. Terminal hardware support

The terminal handler allows a user to access basic hardware related functions; e.g. line speed, modem control, parity, and stop bits. A special signal, SIGHUP, is automatically sent to processes in a terminal's process group when a carrier transition is detected. This is normally associated with a user hanging up on a modem controlled terminal line.

2.4.2. Structured devices

Structures devices are typified by disks and magnetic tapes, but may represent any random-access device. The system performs read-modify-write type buffering actions on block devices to allow them to be read and written in a totally random access fashion like ordinary files. File systems are normally created in block devices.

2.4.3. Unstructured devices

Unstructured devices are those devices which do not support block structure. Familiar unstructured devices are raw communications lines (with no terminal handler), raster plotters, magnetic tape and disks unfettered by buffering and permitting large block input/output and positioning and formatting commands.

2.5. Process and kernel descriptors

The status of the facilities in this section is still under discussion. The *ptrace* facility of 4.1BSD is provided in 4.2BSD. Planned enhancements would allow a descriptor based process control facility.

I. Summary of facilities

1. Kernel primitives

1.1. Process naming and protection

sethostid	set UNIX host id
gethostid	get UNIX host id
sethostname	set UNIX host name
gethostname	get UNIX host name
getpid	get process id
fork	create new process
exit	terminate a process
execve	execute a different process
getuid	get user id
geteuid	get effective user id
setreuid	set real and effective user id's
getgid	get accounting group id
getegid	get effective accounting group id
getgroups	get access group set
setregid	set real and effective group id's
setgroups	set access group set
getpgrp	get process group
setpgrp	set process group

1.2 Memory management

<mman.h>	memory management definitions
sbrk	change data section size
sstk†	change stack section size
getpagesize	get memory page size
mmap†	map pages of memory
mremap†	remap pages in memory
munmap†	unmap memory
mprotect†	change protection of pages
madvise†	give memory management advice
mincore†	determine core residency of pages

1.3 Signals

<signal.h>	signal definitions
sigvec	set handler for signal
kill	send signal to process
killpgrp	send signal to process group
sigblock	block set of signals
sigsetmask	restore set of blocked signals
sigpause	wait for signals
sigstack	set software stack for signals

1.4 Timing and statistics

<sys/time.h>	time-related definitions
gettimeofday	get current time and timezone
settimeofday	set current time and timezone
getitimer	read an interval timer
setitimer	get and set an interval timer

† Not supported in 4.2BSD.

profil	profile process

1.5 Descriptors

getdtablesize	descriptor reference table size
dup	duplicate descriptor
dup2	duplicate to specified index
close	close descriptor
select	multiplex input/output
fcntl	control descriptor options
wrap†	wrap descriptor with protocol

1.6 Resource controls

<sys/resource.h>	resource-related definitions
getpriority	get process priority
setpriority	set process priority
getrusage	get resource usage
getrlimit	get resource limitations
setrlimit	set resource limitations

1.7 System operation support

mount	mount a device file system
swapon	add a swap device
umount	umount a file system
sync	flush system caches
reboot	reboot a machine
acct	specify accounting file

2. System facilities

2.1 Generic operations

read	read data
write	write data
<sys/uio.h>	scatter-gather related definitions
readv	scattered data input
writev	gathered data output
<sys/ioctl.h>	standard control operations
ioctl	device control operation

2.2 File system

Operations marked with a * exist in two forms: as shown, operating on a file name, and operating on a file descriptor, when the name is preceded with a "f".

<sys/file.h>	file system definitions
chdir	change directory
chroot	change root directory
mkdir	make a directory
rmdir	remove a directory
open	open a new or existing file
mknod	make a special file
portal†	make a portal entry
unlink	remove a link
stat*	return status for a file

† Not supported in 4.2BSD.

lstat	returned status of link
chown*	change owner
chmod*	change mode
utimes	change access/modify times
link	make a hard link
symlink	make a symbolic link
readlink	read contents of symbolic link
rename	change name of file
lseek	reposition within file
truncate*	truncate file
access	determine accessibility
flock	lock a file

2.3 Communications

<sys/socket.h>	standard definitions
socket	create socket
bind	bind socket to name
getsockname	get socket name
listen	allow queueing of connections
accept	accept a connection
connect	connect to peer socket
socketpair	create pair of connected sockets
sendto	send data to named socket
send	send data to connected socket
recvfrom	receive data on unconnected socket
recv	receive data on connected socket
sendmsg	send gathered data and/or rights
recvmsg	receive scattered data and/or rights
shutdown	partially close full-duplex connection
getsockopt	get socket option
setsockopt	set socket option

2.5 Terminals, block and character devices

2.4 Processes and kernel hooks