4.4BSD
Programmer's
Supplementary Documents
(PSD)

Now in its twentieth year, the USENIX Association, the UNIX and Advanced Computing Systems professional and technical organization, is a not-for-profit membership association of individuals and institutions with an interest in UNIX and UNIX-like systems, and, by extension, C++, X windows, and other advanced tools and technologies.

USENIX and its members are dedicated to:

- fostering innovation and communicating research and technological developments,
- sharing ideas and experience relevant to UNIX, UNIX-related, and advanced computing systems, and
- providing a neutral forum for the exercise of critical thought and airing of technical issues.

USENIX publishes a journal (**Computing Systems**), a newsletter (*;login:*), Proceedings from its frequent Conferences and Symposia, and a Book Series.

SAGE, The Systems Administrators Guild, a Special Technical Group with the USENIX Association, is dedicated to the advancement of system administration as a profession.

SAGE brings together systems managers and administrators to:

- propagate knowledge of good professional practice,
- recruit talented individuals to the profession,
- recognize individuals who attain professional excellence,
- foster technical development and share solutions to technical problems, and
- communicate in an organized voice with users, management, and vendors on system administration topics.

4.4BSD Programmer's Supplementary Documents (PSD)

Berkeley Software Distribution

April, 1994

Computer Systems Research Group
University of California at Berkeley

A USENIX Association Book
O'Reilly & Associates, Inc.
103 Morris Street, Suite A
Sebastopol, CA 94572

First Printing, 1994
Second Printing, 1995

Copyright 1979, 1980, 1983, 1986, 1988, 1989, 1991, 1992, 1993, 1994 The Regents of the University of California. All rights reserved.

Other than the specific manual pages and documents listed below as copyrighted by AT&T, redistribution and use of this manual in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1) Redistributions of this manual must retain the copyright notices on this page, this list of conditions and the following disclaimer.

2) Software or documentation that incorporates part of this manual must reproduce the copyright notices on this page, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3) All advertising materials mentioning features or use of this software must display the following acknowledgement: ''This product includes software developed by the University of California, Berkeley and its contributors.''

4) Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Documents PSD:1, 2, 3, 4, 6, 11, 15, 16, and 17 are copyright 1979, AT&T Bell Laboratories, Incorporated. Document PSD:8 is a modification of an earlier document that is copyrighted 1979 by AT&T Bell Laboratories, Incorporated. Holders of UNIX™/32V, System III, or System V software licenses 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.

Document PSD:10 is part of the user contributed software and is copyright 1992 by the Free Software Foundation, Inc. Permission is granted to make and distribute verbatim copies of this document provided the copyright notice and this permission notice are preserved on all copies.

Document PSD:13 is part of the user contributed software and is copyright 1983 by Walter F. Tichy. Permission to copy the RCS documentation or any portion thereof as necessary for licensed use of the software is granted to licensees of this software, provided this copyright notice is included.

The views and conclusions contained in this manual are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Regents of the University of California.

The 4.4BSD Daemon used on the cover is copyright 1994 by Marshall Kirk McKusick and is reproduced with permission.
This book was printed and bound in the United States of America.
Distributed by O'Reilly & Associates, Inc.

[recycle logo] This book is printed on acid-free paper with 50% recycled content, 10-13% post-consumer waste. O'Reilly & Associates is committed to using paper with the highest recycled content available consistent with high quality.

ISBN: 1-56592-079-1

Contents

Introduction ix

Documents of Historical Interest

The Unix Time–Sharing System tabbed 1

Dennis Ritchie and Ken Thompson's original paper about UNIX, reprinted from Communications of the ACM.

Unix Implementation tabbed 2

Ken Thompson's description of the implementation of the Version 7 kernel and file system.

The Unix I/O System tabbed 3

Dennis Ritchie's overview of the I/O System of Version 7; still helpful for those writing device drivers.

Unix Programming – Second Edition tabbed 4

Describes the programming interface to the UNIX version 7 operating system and the standard I/O library. Should be supplemented by Kernighan and Pike, ''The UNIX Programming Environment'', Prentice-Hall, 1984 and especially by the Programmer Reference Manual section 2 (system calls) and 3 (library routines).

Berkeley Software Architecture Manual (4.4 Edition) tabbed 5

A concise and terse description of the system call interface provided in Berkeley Unix, as revised for 4.4BSD. This will never be a best seller.

Languages in common use

The C Programming Language – Reference Manual tabbed 6

Official statement of the syntax of C. Should be supplemented by ''The C Programming Language,'' B.W. Kernighan and D.M. Ritchie, Prentice-Hall, 1978, that contains a tutorial introduction and many examples.

Berkeley Pascal User´s Manual tabbed 7

An implementation of this language popular for learning to program.

A Portable Fortran 77 Compiler tabbed 8

A revised version of the document which originally appeared in Volume 2b of the Bell Labs documentation; this version reflects the work done at Berkeley.

Introduction to the f77 I/O Library tabbed 9

A description of the revised input/output library for Fortran 77, reflecting work carried out at Berkeley.

Programming Tools

Debugging with GDB: The GNU Source-Level Debugger tabbed 10

How to debug programs using the source level *gdb* debugger (or how to debug programs without having to know much about machine language).

A Tutorial Introduction to ADB tabbed 11

How to debug programs using the assembly-language level *adb* debugger.

Make – A Program for Maintaining Computer Programs tabbed 12

Indispensable tool for making sure large programs are properly compiled with minimal effort.

An Introduction to the Revision Control System tabbed 13

RCS is a user-contributed tool for working together with other people without stepping on each other's toes. An alternative to *sccs* for controlling software changes.

An Introduction to the Source Code Control System tabbed 14

A useful introductory article for those users with installations licensed for SCCS.

YACC: Yet Another Compiler-Compiler tabbed 15

Converts a BNF specification of a language and semantic actions written in C into a compiler for that language.

LEX – A Lexical Analyzer Generator tabbed 16

Creates a recognizer for a set of regular expressions: each regular expression can be followed by arbitrary C code to be executed upon finding the regular expression.

The M4 Macro Processor tabbed 17

M4 is a macro processor useful in its own right and as a front-end for C, Ratfor, and Cobol.

gprof: a Call Graph Execution Profiler tabbed 18

A program to show the call graph and execution time of a program. Indispensable aid for improving the running time of almost everything.

Programming Libraries

Screen Updating and Cursor Movement Optimization tabbed 19

Describes the *curses* package, an aid for writing screen-oriented, terminal-independent programs.

General Reference

An Introductory 4.4BSD Interprocess Communication Tutorial tabbed 20

How to write programs that use the Interprocess Communication Facilities of 4.4BSD.

An Advanced 4.4BSD Interprocess Communication Tutorial tabbed 21

The reference document (with some examples) for the Interprocess Communication Facilities of 4.4BSD.

List of Documents inside back cover

Introduction

The documentation for 4.4BSD is in a format similar to the one used for the 4.2BSD and 4.3BSD manuals. It is divided into three sets; each set consists of one or more volumes. The abbreviations for the volume names are listed in square brackets; the abbreviations for the manual sections are listed in parenthesis.

I. User's Documents
- User's Reference Manual [URM]
  - Commands (1)
  - Games (6)
  - Macro packages and language conventions (7)
- User's Supplementary Documents [USD]
  - Getting Started
  - Basic Utilities
  - Communicating with the World
  - Text Editing
  - Document Preparation
  - Amusements

II. Programmer's Documents
- Programmer's Reference Manual [PRM]
  - System calls (2)
  - Subroutines (3)
  - Special files (4)
  - File formats and conventions (5)
- Programmer's Supplementary Documents [PSD]
  - Documents of Historic Interest
  - Languages in common use
  - Programming Tools
  - Programming Libraries
  - General Reference

III. System Manager's Manual [SMM]
- Maintenance commands (8)
- System Installation and Administration

References to individual documents are given as ''volume:document'', thus USD:1 refers to the first document in the ''User's Supplementary Documents''. References to manual pages are given as ''*name*(section)'' thus *sh*(1) refers to the shell manual entry in section 1.

The manual pages give descriptions of the features of the 4.4BSD system, as developed at the University of California at Berkeley. They do not attempt to provide perspective or tutorial information about the 4.4BSD operating system, its facilities, or its implementation. Various documents on those topics are contained in the ''UNIX User's Supplementary Documents'' (USD), the ''UNIX Programmer's Supplementary Documents'' (PSD), and ''UNIX System Manager's Manual'' (SMM). In particular, for an overview see ''The UNIX Time-Sharing System'' (PSD:1) by Ritchie and Thompson; for a tutorial see ''UNIX for Beginners'' (USD:1) by Kernighan, and for an guide to the new features of this latest version, see ''Berkeley Software Architecture Manual (4.4 Edition)'' (PSD:5).

Within the area it surveys, this volume attempts to be timely, complete and concise. Where the latter two objectives conflict, the obvious is often left unsaid in favor of brevity. It is intended that each program be described as it is, not as it should be. Inevitably, this means that various sections will soon be out of date.

Commands are programs intended to be invoked directly by the user, in contrast to subroutines, that are intended to be called by the user's programs. User commands are described in URM section 1. Commands generally

reside in directory */bin* (for *bin*ary programs). Some programs also reside in */usr/bin,* to save space in */bin.* These directories are searched automatically by the command interpreters. Additional directories that may be of interest include */usr/contrib/bin,* which has contributed software */usr/old/bin,* which has old but sometimes still useful software and */usr/local/bin,* which contains software local to your site.

Games have been relegated to URM section 6 and */usr/games,* to keep them from contaminating the more staid information of URM section 1.

Miscellaneous collection of information necessary for writing in various specialized languages such as character codes, macro packages for typesetting, etc is contained in URM section 7.

System calls are entries into the BSD kernel. The system call interface is identical to a C language procedure call; the equivalent C procedures are described in PRM section 2.

An assortment of subroutines is available; they are described in PRM section 3. The primary libraries in which they are kept are described in *intro*(3). The functions are described in terms of C.

PRM section 4 discusses the characteristics of each system ‘‘file’’ that refers to an I/O device. The names in this section refer to the HP300 device names for the hardware, instead of the names of the special files themselves.

The file formats and conventions (PRM section 5) documents the structure of particular kinds of files; for example, the form of the output of the loader and assembler is given. Excluded are files used by only one command, for example the assembler's intermediate files.

Commands and procedures intended for use primarily by the system administrator are described in SMM section 8. The files described here are almost all kept in the directory */etc.* The system administration binaries reside in */sbin,* and */usr/sbin.*

Each section consists of independent entries of a page or so each. The name of the entry is in the upper corners of its pages, together with the section number. Entries within each section are alphabetized. The page numbers of each entry start at 1; it is infeasible to number consecutively the pages of a document like this that is republished in many variant forms.

All entries are based on a common format; not all subsections always appear.

The *name* subsection lists the exact names of the commands and subroutines covered under the entry and gives a short description of their purpose.

The *synopsis* summarizes the use of the program being described. A few conventions are used, particularly in the Commands subsection:

Boldface words are considered literals, and are typed just as they appear.

Square brackets [ ] around an argument show that the argument is optional. When an argument is given as ‘‘name’’, it always refers to a file name.

Ellipses ‘‘...’’ are used to show that the previous argument-prototype may be repeated.

A final convention is used by the commands themselves. An argument beginning with a minus sign ‘‘–’’ usually means that it is an option-specifying argument, even if it appears in a position where a file name could appear. Therefore, it is unwise to have files whose names begin with ‘‘–’’.

The *description* subsection discusses in detail the subject at hand.

The *files* subsection gives the names of files that are built into the program.

A *see also* subsection gives pointers to related information.

A *diagnostics* subsection discusses the diagnostic indications that may be produced. Messages that are intended to be self-explanatory are not listed.

The *bugs* subsection gives known bugs and sometimes deficiencies. Occasionally the suggested fix is also described.

At the beginning of URM, PRM, and SSM is a List of Manual Pages, organized by section and alphabetically within each section, and a Permuted Index derived from that List. Within each index entry, the title of the writeup to which it refers is followed by the appropriate section number in parentheses. This fact is important because there is considerable name duplication among the sections, arising principally from commands that exist only to exercise a particular system call. Finally, there is a list of documents on the inside back cover of each

volume.

The UNIX Time-Sharing System*

D. M. Ritchie and K. Thompson

ABSTRACT

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

i A hierarchical file system incorporating demountable volumes,

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

iii The ability to initiate asynchronous processes,

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

v Over 100 subsystems including a dozen languages,

vi High degree of portability.

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

1. INTRODUCTION

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

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

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

---

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

† UNIX is a registered trademark of The Open Group in the U.S. and other countries.

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

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

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

II. HARDWARE AND SOFTWARE ENVIRONMENT

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

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

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

III. THE FILE SYSTEM

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

3.1 Ordinary files

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

3.2 Directories

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

system controls the contents of directories. However, anyone with appropriate permission may read a directory just like any other file.

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

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

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

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

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

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

3.3 Special files

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

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

3.4 Removable file systems

Although the root of the file system is always stored on the same device, it is not necessary that the entire file system hierarchy reside on this device. There is a **mount** system request with two arguments: the name of an existing ordinary file, and the name of a special file whose associated storage volume (e.g., a disk pack) should have the structure of an independent file system containing its own

directory hierarchy. The effect of **mount** is to cause references to the heretofore ordinary file to refer instead to the root directory of the file system on the removable volume. In effect, **mount** replaces a leaf of the hierarchy tree (the ordinary file) by a whole new subtree (the hierarchy stored on the removable volume). After the **mount**, there is virtually no distinction between files on the removable volume and those in the permanent file system. In our installation, for example, the root directory resides on a small partition of one of our disk drives, while the other drive, which contains the user's files, is mounted by the system initialization sequence. A mountable file system is generated by writing on its corresponding special file. A utility program is available to create an empty file system, or one may simply copy an existing file system.

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

3.5 Protection

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

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

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

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

3.6 I/O calls

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

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

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

filep = open (name, flag)

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

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

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

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

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

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

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

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

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

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

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

location = lseek (filep, offset, base)

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

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

owner of a file, create a directory, make a link to an existing file, delete a file.

IV. IMPLEMENTATION OF THE FILE SYSTEM

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

i the user and group-ID of its owner

ii its protection bits

iii the physical disk or tape addresses for the file contents

iv its size

v time of creation, last use, and last modification

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

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

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

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

The space on all disks that contain a file system is divided into a number of 512-byte blocks logically addressed from 0 up to a limit that depends on the device. There is space in the i-node of each file for 13 device addresses. For nonspecial files, the first 10 device addresses point at the first 10 blocks of the file. If the file is larger than 10 blocks, the 11 device address points to an indirect block containing up to 128 addresses of additional blocks in the file. Still larger files use the twelfth device address of the i-node to point to a double-indirect block naming 128 indirect blocks, each pointing to 128 blocks of the file. If required, the thirteenth device address is a triple-indirect block. Thus files may conceptually grow to $[(10+128+128^2+128^3)\cdot 512]$ bytes. Once opened, bytes numbered below 5120 can be read with a single disk access; bytes in the range 5120 to 70,656 require two accesses; bytes in the range 70,656 to 8,459,264 require three accesses; bytes from there to the largest file (1,082,201,088) require four accesses. In practice, a device cache mechanism (see below) proves effective in eliminating most of the indirect fetches.

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

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

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

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

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

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

V. PROCESSES AND IMAGES

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

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

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

5.1 Processes

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

processid = fork( )

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

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

5.2 Pipes

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

filep = pipe( )

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

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

5.3 Execution of programs

Another major system primitive is invoked by

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

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

5.4 Process synchronization

Another process control system call:

processid = wait(status)

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

5.5 Termination

Lastly:

exit(status)

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

VI. THE SHELL

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

command arg_1 arg_2 ... arg_n

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

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

6.1 Standard I/O

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

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

```
ls
```

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

```
ls >there
```

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

```
ed
```

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

```
ed <script
```

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

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

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

6.2 Filters

An extension of the standard I/O notion is used to direct output from one command to the input of another. A sequence of commands separated by vertical bars causes the shell to execute all the

commands simultaneously and to arrange that the standard output of each command be delivered to the standard input of the next command in the sequence. Thus in the command line:

```
ls | pr −2 | opr
```

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

This procedure could have been carried out more clumsily by:

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

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

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

6.3 Command separators; multitasking

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

```
ls; ed
```

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

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

```
as source >output &
```

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

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

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

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

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

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

6.4 The shell as a command; command files

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

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

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

```
sh <tryout
```

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

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

6.5 Implementation of the shell

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

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

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

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

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

```
sh <comfile
```

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

6.6 Initialization

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

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

6.7 Other programs as shell

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

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

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

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

VII. TRAPS

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

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

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

without floating-point hardware, unimplemented instructions are caught and floating-point instructions are interpreted.

VIII. PERSPECTIVE

Perhaps paradoxically, the success of the UNIX system is largely due to the fact that it was not designed to meet any predefined objectives. The first version was written when one of us (Thompson), dissatisfied with the available computer facilities, discovered a little-used PDP-7 and set out to create a more hospitable environment. This (essentially personal) effort was sufficiently successful to gain the interest of the other author and several colleagues, and later to justify the acquisition of the PDP-11/20, specifically to support a text editing and formatting system. When in turn the 11/20 was outgrown, the system had proved useful enough to persuade management to invest in the PDP-11/45, and later in the PDP-11/70 and Interdata 8/32 machines, upon which it developed to its present form. Our goals throughout the effort, when articulated at all, have always been to build a comfortable relationship with the machine and to explore ideas and inventions in operating systems and other software. We have not been faced with the need to satisfy someone else's requirements, and for this freedom we are grateful.

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

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

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

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

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

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

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

Likewise, the process-control scheme and the command interface have proved both convenient and efficient. Because the shell operates as an ordinary, swappable user program, it consumes no "wired-down" space in the system proper, and it may be made as powerful as desired at little cost. In particular, given the framework in which the shell executes as a process that spawns other processes to perform

commands, the notions of I/O redirection, background processes, command files, and user-selectable system interfaces all become essentially trivial to implement.

Influences

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

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

IX. STATISTICS

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

Overall, we have today:

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

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

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

X. ACKNOWLEDGMENTS

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

References

1. L. P. Deutsch and B. W. Lampson, “An online editor,” *Comm. Assoc. Comp. Mach.*, vol. 10, no. 12, pp. 793-799, 803, December 1967.
2. B. W. Kernighan and L. L. Cherry, “A System for Typesetting Mathematics,” *Comm. Assoc. Comp. Mach.*, vol. 18, pp. 151-157, Bell Laboratories, Murray Hill, New Jersey, March 1975.
3. This issue, B. W. Kernighan, M. E. Lesk, and J. F. Ossanna, “UNIX Time-Sharing System: Document Preparation,” *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 2115-2135, 1978.

4. T. A. Dolotta and J. R. Mashey, "An Introduction to the Programmer's Workbench," *Proc. 2nd Int. Conf. on Software Engineering*, pp. 164-168, October 13-15, 1976.

5. This issue, T. A. Dolotta, R. C. Haight, and J. R. Mashey, "UNIX Time-Sharing System: The Programmer's Workbench," *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 2177-2200, 1978.

6. This issue, H. Lycklama, "UNIX Time-Sharing System: UNIX on a Microprocessor," *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 2087-2101, 1978.

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

8. Aleph-null, "Computer Recreations," *Software Practice and Experience*, vol. 1, no. 2, pp. 201-204, April-June 1971.

9. This issue, S. R. Bourne, "UNIX Time-Sharing System: The UNIX Shell," *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 1971-1990, 1978.

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

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

12. D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, "TENEX, a Paged Time Sharing System for the PDP-10," *Comm. Assoc. Comp. Mach.*, vol. 15, no. 3, pp. 135-143, March 1972.

UNIX Implementation

K. Thompson

AT&T Bell Laboratories
Murray Hill, NJ

ABSTRACT

This paper describes in high-level terms the implementation of the resident UNIX† kernel. This discussion is broken into three parts. The first part describes how the UNIX system views processes, users, and programs. The second part describes the I/O system. The last part describes the UNIX file system.

1. INTRODUCTION

The UNIX kernel consists of about 10,000 lines of C code and about 1,000 lines of assembly code. The assembly code can be further broken down into 200 lines included for the sake of efficiency (they could have been written in C) and 800 lines to perform hardware functions not possible in C.

This code represents 5 to 10 percent of what has been lumped into the broad expression "the UNIX operating system." The kernel is the only UNIX code that cannot be substituted by a user to his own liking. For this reason, the kernel should make as few real decisions as possible. This does not mean to allow the user a million options to do the same thing. Rather, it means to allow only one way to do one thing, but have that way be the least-common divisor of all the options that might have been provided.

What is or is not implemented in the kernel represents both a great responsibility and a great power. It is a soap-box platform on "the way things should be done." Even so, if "the way" is too radical, no one will follow it. Every important decision was weighed carefully. Throughout, simplicity has been substituted for efficiency. Complex algorithms are used only if their complexity can be localized.

2. PROCESS CONTROL

In the UNIX system, a user executes programs in an environment called a user process. When a system function is required, the user process calls the system as a subroutine. At some point in this call, there is a distinct switch of environments. After this, the process is said to be a system process. In the normal definition of processes, the user and system processes are different phases of the same process (they never execute simultaneously). For protection, each system process has its own stack.

The user process may execute from a read-only text segment, which is shared by all processes executing the same code. There is no *functional* benefit from shared-text segments. An *efficiency* benefit comes from the fact that there is no need to swap read-only segments out because the original copy on secondary memory is still current. This is a great benefit to interactive programs that tend to be swapped while waiting for terminal input. Furthermore, if two processes are executing simultaneously from the same copy of a read-only segment, only one copy needs to reside in primary memory. This is a secondary effect, because simultaneous execution of a program is not common. It is ironic that this effect, which reduces the use of primary memory, only comes into play when there is an overabundance of primary memory, that is, when there is enough memory to keep waiting processes loaded.

---

† UNIX is a registered trademark of The Open Group in the U.S. and other countries.

All current read-only text segments in the system are maintained from the *text table*. A text table entry holds the location of the text segment on secondary memory. If the segment is loaded, that table also holds the primary memory location and the count of the number of processes sharing this entry. When this count is reduced to zero, the entry is freed along with any primary and secondary memory holding the segment. When a process first executes a shared-text segment, a text table entry is allocated and the segment is loaded onto secondary memory. If a second process executes a text segment that is already allocated, the entry reference count is simply incremented.

A user process has some strictly private read-write data contained in its data segment. As far as possible, the system does not use the user's data segment to hold system data. In particular, there are no I/O buffers in the user address space.

The user data segment has two growing boundaries. One, increased automatically by the system as a result of memory faults, is used for a stack. The second boundary is only grown (or shrunk) by explicit requests. The contents of newly allocated primary memory is initialized to zero.

Also associated and swapped with a process is a small fixed-size system data segment. This segment contains all the data about the process that the system needs only when the process is active. Examples of the kind of data contained in the system data segment are: saved central processor registers, open file descriptors, accounting information, scratch data area, and the stack for the system phase of the process. The system data segment is not addressable from the user process and is therefore protected.

Last, there is a process table with one entry per process. This entry contains all the data needed by the system when the process is *not* active. Examples are the process's name, the location of the other segments, and scheduling information. The process table entry is allocated when the process is created, and freed when the process terminates. This process entry is always directly addressable by the kernel.

Figure 1 shows the relationships between the various process control data. In a sense, the process table is the definition of all processes, because all the data associated with a process may be accessed starting from the process table entry.

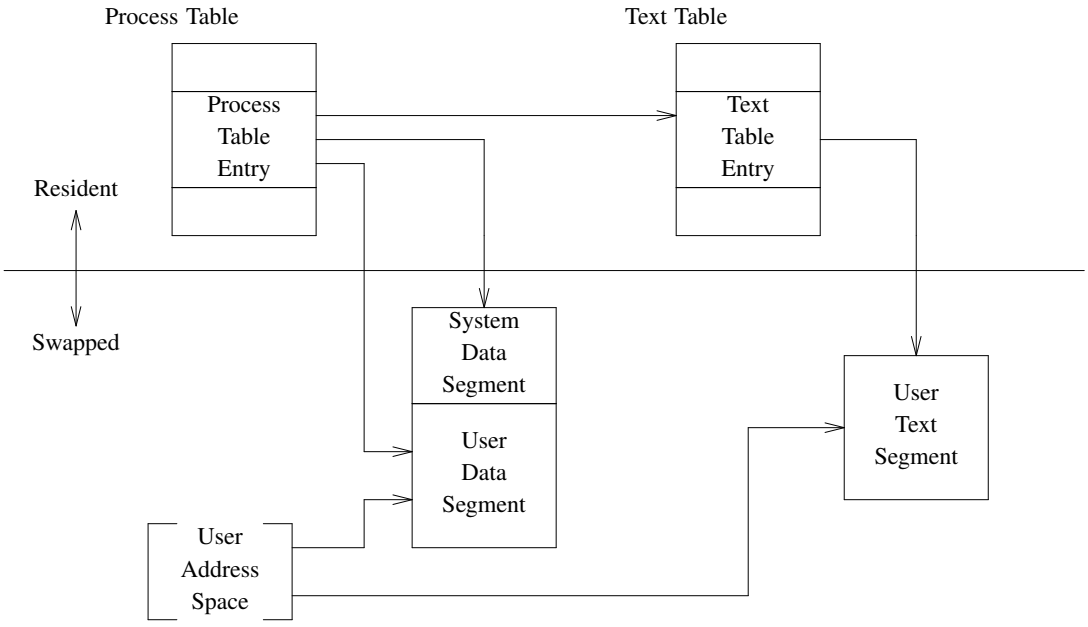


Fig. 1—Process control data structure.

2.1. Process creation and program execution

Processes are created by the system primitive **fork**. The newly created process (child) is a copy of the original process (parent). There is no detectable sharing of primary memory between the two processes. (Of course, if the parent process was executing from a read-only text segment, the child will share the text segment.) Copies of all writable data segments are made for the child process. Files that were open before the **fork** are truly shared after the **fork**. The processes are informed as to their part in the relationship to allow them to select their own (usually non-identical) destiny. The parent may **wait** for the termination of any of its children.

A process may **exec** a file. This consists of exchanging the current text and data segments of the process for new text and data segments specified in the file. The old segments are lost. Doing an **exec** does *not* change processes; the process that did the **exec** persists, but after the **exec** it is executing a different program. Files that were open before the **exec** remain open after the **exec**.

If a program, say the first pass of a compiler, wishes to overlay itself with another program, say the second pass, then it simply **exec**s the second program. This is analogous to a "goto." If a program wishes to regain control after **exec**ing a second program, it should **fork** a child process, have the child **exec** the second program, and have the parent **wait** for the child. This is analogous to a "call." Breaking up the call into a binding followed by a transfer is similar to the subroutine linkage in SL-5.[1]

2.2. Swapping

The major data associated with a process (the user data segment, the system data segment, and the text segment) are swapped to and from secondary memory, as needed. The user data segment and the system data segment are kept in contiguous primary memory to reduce swapping latency. (When low-latency devices, such as bubbles, CCDs, or scatter/gather devices, are used, this decision will have to be reconsidered.) Allocation of both primary and secondary memory is performed by the same simple first-fit algorithm. When a process grows, a new piece of primary memory is allocated. The contents of the old memory is copied to the new memory. The old memory is freed and the tables are updated. If there is not enough primary memory, secondary memory is allocated instead. The process is swapped out onto the secondary memory, ready to be swapped in with its new size.

One separate process in the kernel, the swapping process, simply swaps the other processes in and out of primary memory. It examines the process table looking for a process that is swapped out and is ready to run. It allocates primary memory for that process and reads its segments into primary memory, where that process competes for the central processor with other loaded processes. If no primary memory is available, the swapping process makes memory available by examining the process table for processes that can be swapped out. It selects a process to swap out, writes it to secondary memory, frees the primary memory, and then goes back to look for a process to swap in.

Thus there are two specific algorithms to the swapping process. Which of the possibly many processes that are swapped out is to be swapped in? This is decided by secondary storage residence time. The one with the longest time out is swapped in first. There is a slight penalty for larger processes. Which of the possibly many processes that are loaded is to be swapped out? Processes that are waiting for slow events (i.e., not currently running or waiting for disk I/O) are picked first, by age in primary memory, again with size penalties. The other processes are examined by the same age algorithm, but are not taken out unless they are at least of some age. This adds hysteresis to the swapping and prevents total thrashing.

These swapping algorithms are the most suspect in the system. With limited primary memory, these algorithms cause total swapping. This is not bad in itself, because the swapping does not impact the execution of the resident processes. However, if the swapping device must also be used for file storage, the swapping traffic severely impacts the file system traffic. It is exactly these small systems that tend to double usage of limited disk resources.

2.3. Synchronization and scheduling

Process synchronization is accomplished by having processes wait for events. Events are represented by arbitrary integers. By convention, events are chosen to be addresses of tables associated with those events. For example, a process that is waiting for any of its children to terminate will wait for an event that is the address of its own process table entry. When a process terminates, it signals the event represented by its parent's process table entry. Signaling an event on which no process is waiting has no effect. Similarly, signaling an event on which many processes are waiting will wake all of them up. This differs considerably from Dijkstra's P and V synchronization operations,[2] in that no memory is associated with events. Thus there need be no allocation of events prior to their use. Events exist simply by being used.

On the negative side, because there is no memory associated with events, no notion of ''how much'' can be signaled via the event mechanism. For example, processes that want memory might wait on an event associated with memory allocation. When any amount of memory becomes available, the event would be signaled. All the competing processes would then wake up to fight over the new memory. (In reality, the swapping process is the only process that waits for primary memory to become available.)

If an event occurs between the time a process decides to wait for that event and the time that process enters the wait state, then the process will wait on an event that has already happened (and may never happen again). This race condition happens because there is no memory associated with the event to indicate that the event has occurred; the only action of an event is to change a set of processes from wait state to run state. This problem is relieved largely by the fact that process switching can only occur in the kernel by explicit calls to the event-wait mechanism. If the event in question is signaled by another process, then there is no problem. But if the event is signaled by a hardware interrupt, then special care must be taken. These synchronization races pose the biggest problem when UNIX is adapted to multiple-processor configurations.[3]

The event-wait code in the kernel is like a co-routine linkage. At any time, all but one of the processes has called event-wait. The remaining process is the one currently executing. When it calls event-wait, a process whose event has been signaled is selected and that process returns from its call to event-wait.

Which of the runable processes is to run next? Associated with each process is a priority. The priority of a system process is assigned by the code issuing the wait on an event. This is roughly equivalent to the response that one would expect on such an event. Disk events have high priority, teletype events are low, and time-of-day events are very low. (From observation, the difference in system process priorities has little or no performance impact.) All user-process priorities are lower than the lowest system priority. User-process priorities are assigned by an algorithm based on the recent ratio of the amount of compute time to real time consumed by the process. A process that has used a lot of compute time in the last real-time unit is assigned a low user priority. Because interactive processes are characterized by low ratios of compute to real time, interactive response is maintained without any special arrangements.

The scheduling algorithm simply picks the process with the highest priority, thus picking all system processes first and user processes second. The compute-to-real-time ratio is updated every second. Thus, all other things being equal, looping user processes will be scheduled round-robin with a 1-second quantum. A high-priority process waking up will preempt a running, low-priority process. The scheduling algorithm has a very desirable negative feedback character. If a process uses its high priority to hog the computer, its priority will drop. At the same time, if a low-priority process is ignored for a long time, its priority will rise.

3. I/O SYSTEM

The I/O system is broken into two completely separate systems: the block I/O system and the character I/O system. In retrospect, the names should have been ''structured I/O'' and ''unstructured I/O,'' respectively; while the term ''block I/O'' has some meaning, ''character I/O'' is a complete misnomer.

Devices are characterized by a major device number, a minor device number, and a class (block or character). For each class, there is an array of entry points into the device drivers. The major device number is used to index the array when calling the code for a particular device driver. The minor device number is passed to the device driver as an argument. The minor number has no significance other than that attributed to it by the driver. Usually, the driver uses the minor number to access one of several identical physical devices.

The use of the array of entry points (configuration table) as the only connection between the system code and the device drivers is very important. Early versions of the system had a much less formal connection with the drivers, so that it was extremely hard to handcraft differently configured systems. Now it is possible to create new device drivers in an average of a few hours. The configuration table in most cases is created automatically by a program that reads the system's parts list.

3.1. Block I/O system

The model block I/O device consists of randomly addressed, secondary memory blocks of 512 bytes each. The blocks are uniformly addressed 0, 1, ... up to the size of the device. The block device driver has the job of emulating this model on a physical device.

The block I/O devices are accessed through a layer of buffering software. The system maintains a list of buffers (typically between 10 and 70) each assigned a device name and a device address. This buffer pool constitutes a data cache for the block devices. On a read request, the cache is searched for the desired block. If the block is found, the data are made available to the requester without any physical I/O. If the block is not in the cache, the least recently used block in the cache is renamed, the correct device driver is called to fill up the renamed buffer, and then the data are made available. Write requests are handled in an analogous manner. The correct buffer is found and relabeled if necessary. The write is performed simply by marking the buffer as "dirty." The physical I/O is then deferred until the buffer is renamed.

The benefits in reduction of physical I/O of this scheme are substantial, especially considering the file system implementation. There are, however, some drawbacks. The asynchronous nature of the algorithm makes error reporting and meaningful user error handling almost impossible. The cavalier approach to I/O error handling in the UNIX system is partly due to the asynchronous nature of the block I/O system. A second problem is in the delayed writes. If the system stops unexpectedly, it is almost certain that there is a lot of logically complete, but physically incomplete, I/O in the buffers. There is a system primitive to flush all outstanding I/O activity from the buffers. Periodic use of this primitive helps, but does not solve, the problem. Finally, the associativity in the buffers can alter the physical I/O sequence from that of the logical I/O sequence. This means that there are times when data structures on disk are inconsistent, even though the software is careful to perform I/O in the correct order. On non-random devices, notably magnetic tape, the inversions of writes can be disastrous. The problem with magnetic tapes is "cured" by allowing only one outstanding write request per drive.

3.2. Character I/O system

The character I/O system consists of all devices that do not fall into the block I/O model. This includes the "classical" character devices such as communications lines, paper tape, and line printers. It also includes magnetic tape and disks when they are not used in a stereotyped way, for example, 80-byte physical records on tape and track-at-a-time disk copies. In short, the character I/O interface means "everything other than block." I/O requests from the user are sent to the device driver essentially unaltered. The implementation of these requests is, of course, up to the device driver. There are guidelines and conventions to help the implementation of certain types of device drivers.

3.2.1. Disk drivers

Disk drivers are implemented with a queue of transaction records. Each record holds a read/write flag, a primary memory address, a secondary memory address, and a transfer byte count. Swapping is accomplished by passing such a record to the swapping device driver. The block I/O interface is implemented by passing such records with requests to fill and empty system buffers. The character I/O interface to the disk drivers create a transaction record that points directly into the user area. The routine

that creates this record also insures that the user is not swapped during this I/O transaction. Thus by implementing the general disk driver, it is possible to use the disk as a block device, a character device, and a swap device. The only really disk-specific code in normal disk drivers is the pre-sort of transactions to minimize latency for a particular device, and the actual issuing of the I/O request.

3.2.2. Character lists

Real character-oriented devices may be implemented using the common code to handle character lists. A character list is a queue of characters. One routine puts a character on a queue. Another gets a character from a queue. It is also possible to ask how many characters are currently on a queue. Storage for all queues in the system comes from a single common pool. Putting a character on a queue will allocate space from the common pool and link the character onto the data structure defining the queue. Getting a character from a queue returns the corresponding space to the pool.

A typical character-output device (paper tape punch, for example) is implemented by passing characters from the user onto a character queue until some maximum number of characters is on the queue. The I/O is prodded to start as soon as there is anything on the queue and, once started, it is sustained by hardware completion interrupts. Each time there is a completion interrupt, the driver gets the next character from the queue and sends it to the hardware. The number of characters on the queue is checked and, as the count falls through some intermediate level, an event (the queue address) is signaled. The process that is passing characters from the user to the queue can be waiting on the event, and refill the queue to its maximum when the event occurs.

A typical character input device (for example, a paper tape reader) is handled in a very similar manner.

Another class of character devices is the terminals. A terminal is represented by three character queues. There are two input queues (raw and canonical) and an output queue. Characters going to the output of a terminal are handled by common code exactly as described above. The main difference is that there is also code to interpret the output stream as ASCII characters and to perform some translations, e.g., escapes for deficient terminals. Another common aspect of terminals is code to insert real-time delay after certain control characters.

Input on terminals is a little different. Characters are collected from the terminal and placed on a raw input queue. Some device-dependent code conversion and escape interpretation is handled here. When a line is complete in the raw queue, an event is signaled. The code catching this signal then copies a line from the raw queue to a canonical queue performing the character erase and line kill editing. User read requests on terminals can be directed at either the raw or canonical queues.

3.2.3. Other character devices

Finally, there are devices that fit no general category. These devices are set up as character I/O drivers. An example is a driver that reads and writes unmapped primary memory as an I/O device. Some devices are too fast to be treated a character at time, but do not fit the disk I/O mold. Examples are fast communications lines and fast line printers. These devices either have their own buffers or ‘‘borrow’’ block I/O buffers for a while and then give them back.

4. THE FILE SYSTEM

In the UNIX system, a file is a (one-dimensional) array of bytes. No other structure of files is implied by the system. Files are attached anywhere (and possibly multiply) onto a hierarchy of directories. Directories are simply files that users cannot write. For a further discussion of the external view of files and directories, see Ref. 4.

The UNIX file system is a disk data structure accessed completely through the block I/O system. As stated before, the canonical view of a ‘‘disk’’ is a randomly addressable array of 512-byte blocks. A file system breaks the disk into four self-identifying regions. The first block (address 0) is unused by the file system. It is left aside for booting procedures. The second block (address 1) contains the so-called ‘‘super-block.’’ This block, among other things, contains the size of the disk and the boundaries of the other regions. Next comes the i-list, a list of file definitions. Each file definition is a 64-byte

structure, called an i-node. The offset of a particular i-node within the i-list is called its i-number. The combination of device name (major and minor numbers) and i-number serves to uniquely name a particular file. After the i-list, and to the end of the disk, come free storage blocks that are available for the contents of files.

The free space on a disk is maintained by a linked list of available disk blocks. Every block in this chain contains a disk address of the next block in the chain. The remaining space contains the address of up to 50 disk blocks that are also free. Thus with one I/O operation, the system obtains 50 free blocks and a pointer where to find more. The disk allocation algorithms are very straightforward. Since all allocation is in fixed-size blocks and there is strict accounting of space, there is no need to compact or garbage collect. However, as disk space becomes dispersed, latency gradually increases. Some installations choose to occasionally compact disk space to reduce latency.

An i-node contains 13 disk addresses. The first 10 of these addresses point directly at the first 10 blocks of a file. If a file is larger than 10 blocks (5,120 bytes), then the eleventh address points at a block that contains the addresses of the next 128 blocks of the file. If the file is still larger than this (70,656 bytes), then the twelfth block points at up to 128 blocks, each pointing to 128 blocks of the file. Files yet larger (8,459,264 bytes) use the thirteenth address for a "triple indirect" address. The algorithm ends here with the maximum file size of 1,082,201,087 bytes.

A logical directory hierarchy is added to this flat physical structure simply by adding a new type of file, the directory. A directory is accessed exactly as an ordinary file. It contains 16-byte entries consisting of a 14-byte name and an i-number. The root of the hierarchy is at a known i-number (*viz.,* 2). The file system structure allows an arbitrary, directed graph of directories with regular files linked in at arbitrary places in this graph. In fact, very early UNIX systems used such a structure. Administration of such a structure became so chaotic that later systems were restricted to a directory tree. Even now, with regular files linked multiply into arbitrary places in the tree, accounting for space has become a problem. It may become necessary to restrict the entire structure to a tree, and allow a new form of linking that is subservient to the tree structure.

The file system allows easy creation, easy removal, easy random accessing, and very easy space allocation. With most physical addresses confined to a small contiguous section of disk, it is also easy to dump, restore, and check the consistency of the file system. Large files suffer from indirect addressing, but the cache prevents most of the implied physical I/O without adding much execution. The space overhead properties of this scheme are quite good. For example, on one particular file system, there are 25,000 files containing 130M bytes of data-file content. The overhead (i-node, indirect blocks, and last block breakage) is about 11.5M bytes. The directory structure to support these files has about 1,500 directories containing 0.6M bytes of directory content and about 0.5M bytes of overhead in accessing the directories. Added up any way, this comes out to less than a 10 percent overhead for actual stored data. Most systems have this much overhead in padded trailing blanks alone.

4.1. File system implementation

Because the i-node defines a file, the implementation of the file system centers around access to the i-node. The system maintains a table of all active i-nodes. As a new file is accessed, the system locates the corresponding i-node, allocates an i-node table entry, and reads the i-node into primary memory. As in the buffer cache, the table entry is considered to be the current version of the i-node. Modifications to the i-node are made to the table entry. When the last access to the i-node goes away, the table entry is copied back to the secondary store i-list and the table entry is freed.

All I/O operations on files are carried out with the aid of the corresponding i-node table entry. The accessing of a file is a straightforward implementation of the algorithms mentioned previously. The user is not aware of i-nodes and i-numbers. References to the file system are made in terms of path names of the directory tree. Converting a path name into an i-node table entry is also straightforward. Starting at some known i-node (the root or the current directory of some process), the next component of the path name is searched by reading the directory. This gives an i-number and an implied device (that of the directory). Thus the next i-node table entry can be accessed. If that was the last component of the path name, then this i-node is the result. If not, this i-node is the directory needed to look up the next component of the path name, and the algorithm is repeated.

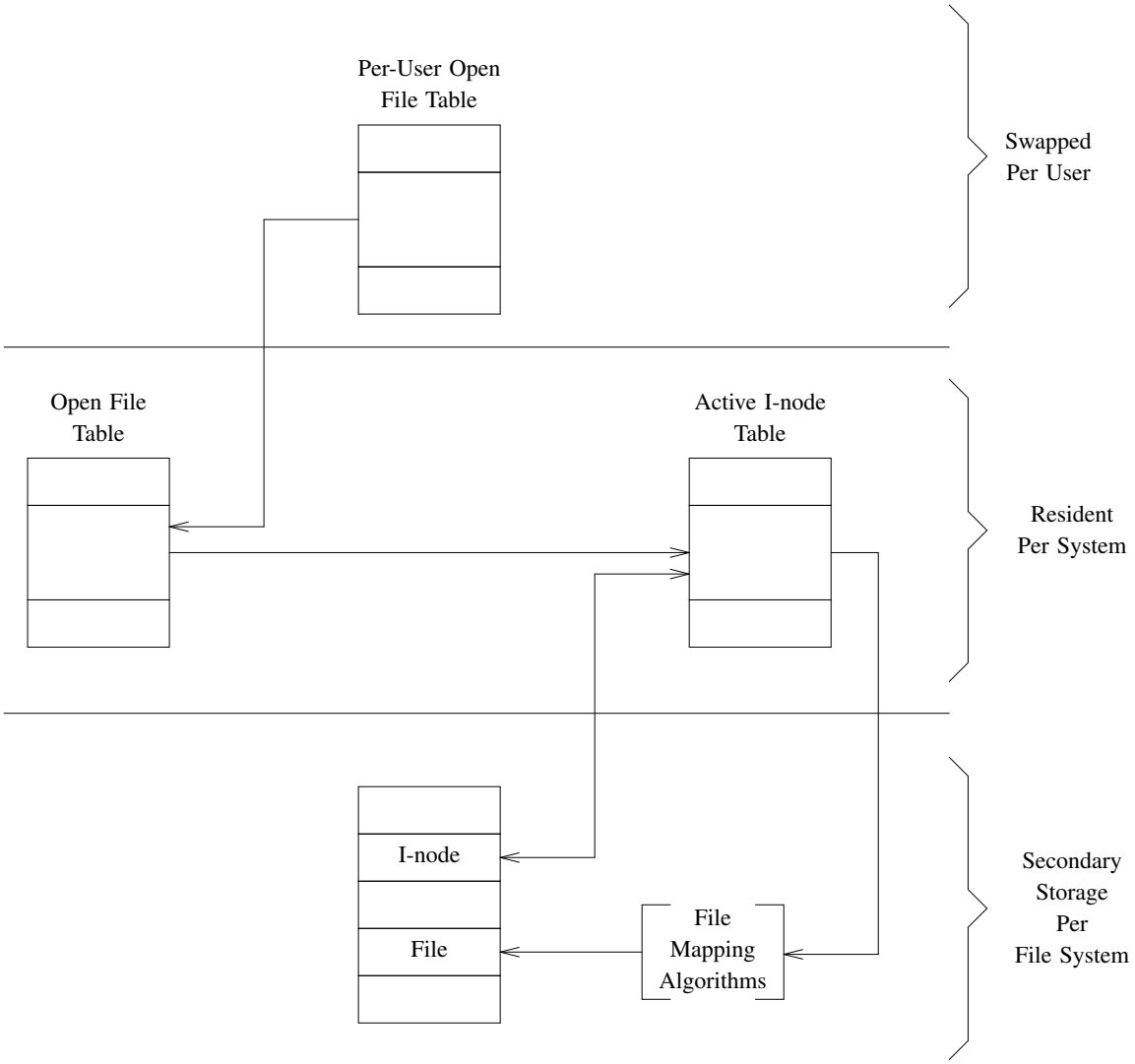


Fig. 2—File system data structure.

The user process accesses the file system with certain primitives. The most common of these are **open**, **create**, **read**, **write**, **seek**, and **close**. The data structures maintained are shown in Fig. 2. In the system data segment associated with a user, there is room for some (usually between 10 and 50) open files. This open file table consists of pointers that can be used to access corresponding i-node table entries. Associated with each of these open files is a current I/O pointer. This is a byte offset of the next read/write operation on the file. The system treats each read/write request as random with an implied seek to the I/O pointer. The user usually thinks of the file as sequential with the I/O pointer automatically counting the number of bytes that have been read/written from the file. The user may, of course, perform random I/O by setting the I/O pointer before reads/writes.

With file sharing, it is necessary to allow related processes to share a common I/O pointer and yet have separate I/O pointers for independent processes that access the same file. With these two conditions, the I/O pointer cannot reside in the i-node table nor can it reside in the list of open files for the process. A new table (the open file table) was invented for the sole purpose of holding the I/O pointer. Processes that share the same open file (the result of **fork**s) share a common open file table entry. A separate open of the same file will only share the i-node table entry, but will have distinct open file table entries.

The main file system primitives are implemented as follows. **open** converts a file system path name into an i-node table entry. A pointer to the i-node table entry is placed in a newly created open file table entry. A pointer to the file table entry is placed in the system data segment for the process. **create** first creates a new i-node entry, writes the i-number into a directory, and then builds the same structure as for an **open**. **read** and **write** just access the i-node entry as described above. **seek** simply manipulates the I/O pointer. No physical seeking is done. **close** just frees the structures built by **open** and **create**. Reference counts are kept on the open file table entries and the i-node table entries to free these structures after the last reference goes away. **unlink** simply decrements the count of the number of directories pointing at the given i-node. When the last reference to an i-node table entry goes away, if the i-node has no directories pointing to it, then the file is removed and the i-node is freed. This delayed removal of files prevents problems arising from removing active files. A file may be removed while still open. The resulting unnamed file vanishes when the file is closed. This is a method of obtaining temporary files.

There is a type of unnamed FIFO file called a **pipe.** Implementation of **pipe**s consists of implied **seek**s before each **read** or **write** in order to implement first-in-first-out. There are also checks and synchronization to prevent the writer from grossly outproducing the reader and to prevent the reader from overtaking the writer.

4.2. Mounted file systems

The file system of a UNIX system starts with some designated block device formatted as described above to contain a hierarchy. The root of this structure is the root of the UNIX file system. A second formatted block device may be mounted at any leaf of the current hierarchy. This logically extends the current hierarchy. The implementation of mounting is trivial. A mount table is maintained containing pairs of designated leaf i-nodes and block devices. When converting a path name into an i-node, a check is made to see if the new i-node is a designated leaf. If it is, the i-node of the root of the block device replaces it.

Allocation of space for a file is taken from the free pool on the device on which the file lives. Thus a file system consisting of many mounted devices does not have a common pool of free secondary storage space. This separation of space on different devices is necessary to allow easy unmounting of a device.

4.3. Other system functions

There are some other things that the system does for the user–a little accounting, a little tracing/debugging, and a little access protection. Most of these things are not very well developed because our use of the system in computing science research does not need them. There are some features that are missed in some applications, for example, better inter-process communication.

The UNIX kernel is an I/O multiplexer more than a complete operating system. This is as it should be. Because of this outlook, many features are found in most other operating systems that are missing from the UNIX kernel. For example, the UNIX kernel does not support file access methods, file disposition, file formats, file maximum size, spooling, command language, logical records, physical records, assignment of logical file names, logical file names, more than one character set, an operator's console, an operator, log-in, or log-out. Many of these things are symptoms rather than features. Many of these things are implemented in user software using the kernel as a tool. A good example of this is the command language.[5] Each user may have his own command language. Maintenance of such code is as easy as maintaining user code. The idea of implementing ''system'' code with general user primitives comes directly from MULTICS.[6]

References

1. R. E. Griswold and D. R. Hanson, "An Overview of SL5," *SIGPLAN Notices*, vol. 12, no. 4, pp. 40-50, April 1977.

2. E. W. Dijkstra, "Cooperating Sequential Processes," in *Programming Languages*, ed. by F. Genuys, pp. 43-112, Academic Press, New York, 1968.

3. J. A. Hawley and W. B. Meyer, "MUNIX, A Multiprocessing Version of UNIX," M.S. Thesis, Naval Postgraduate School, Monterey, Cal., 1975.

4. This issue, D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 1905-1929, 1978.

5. This issue, S. R. Bourne, "UNIX Time-Sharing System: The UNIX Shell," *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 1971-1990, 1978.

6. E. I. Organick, *The MULTICS System,* M.I.T. Press, Cambridge, Mass., 1972.

The UNIX I/O System

Dennis M. Ritchie

AT&T Bell Laboratories
Murray Hill, NJ

This paper gives an overview of the workings of the UNIX† I/O system. It was written with an eye toward providing guidance to writers of device driver routines, and is oriented more toward describing the environment and nature of device drivers than the implementation of that part of the file system which deals with ordinary files.

It is assumed that the reader has a good knowledge of the overall structure of the file system as discussed in the paper "The UNIX Time-sharing System." A more detailed discussion appears in "UNIX Implementation;" the current document restates parts of that one, but is still more detailed. It is most useful in conjunction with a copy of the system code, since it is basically an exegesis of that code.

Device Classes

There are two classes of device: *block* and *character.* The block interface is suitable for devices like disks, tapes, and DECtape which work, or can work, with addressible 512-byte blocks. Ordinary magnetic tape just barely fits in this category, since by use of forward and backward spacing any block can be read, even though blocks can be written only at the end of the tape. Block devices can at least potentially contain a mounted file system. The interface to block devices is very highly structured; the drivers for these devices share a great many routines as well as a pool of buffers.

Character-type devices have a much more straightforward interface, although more work must be done by the driver itself.

Devices of both types are named by a *major* and a *minor* device number. These numbers are generally stored as an integer with the minor device number in the low-order 8 bits and the major device number in the next-higher 8 bits; macros *major* and *minor* are available to access these numbers. The major device number selects which driver will deal with the device; the minor device number is not used by the rest of the system but is passed to the driver at appropriate times. Typically the minor number selects a subdevice attached to a given controller, or one of several similar hardware interfaces.

The major device numbers for block and character devices are used as indices in separate tables; they both start at 0 and therefore overlap.

Overview of I/O

The purpose of the *open* and *creat* system calls is to set up entries in three separate system tables. The first of these is the *u_ofile* table, which is stored in the system's per-process data area *u.* This table is indexed by the file descriptor returned by the *open* or *creat,* and is accessed during a *read, write,* or other operation on the open file. An entry contains only a pointer to the corresponding entry of the *file* table, which is a per-system data base. There is one entry in the *file* table for each instance of *open* or *creat.* This table is per-system because the same instance of an open file must be shared among the several processes which can result from *forks* after the file is opened. A *file* table entry contains flags which indicate whether the file was open for reading or writing or is a pipe, and a count which is used to decide when all processes using the entry have terminated or closed the file (so the entry can be abandoned). There is also a 32-bit file offset which is used to indicate where in the file the

†UNIX is a Trademark of Bell Laboratories.

next read or write will take place. Finally, there is a pointer to the entry for the file in the *inode* table, which contains a copy of the file's i-node.

Certain open files can be designated ''multiplexed'' files, and several other flags apply to such channels. In such a case, instead of an offset, there is a pointer to an associated multiplex channel table. Multiplex channels will not be discussed here.

An entry in the *file* table corresponds precisely to an instance of *open* or *creat;* if the same file is opened several times, it will have several entries in this table. However, there is at most one entry in the *inode* table for a given file. Also, a file may enter the *inode* table not only because it is open, but also because it is the current directory of some process or because it is a special file containing a currently-mounted file system.

An entry in the *inode* table differs somewhat from the corresponding i-node as stored on the disk; the modified and accessed times are not stored, and the entry is augmented by a flag word containing information about the entry, a count used to determine when it may be allowed to disappear, and the device and i-number whence the entry came. Also, the several block numbers that give addressing information for the file are expanded from the 3-byte, compressed format used on the disk to full *long* quantities.

During the processing of an *open* or *creat* call for a special file, the system always calls the device's *open* routine to allow for any special processing required (rewinding a tape, turning on the data-terminal-ready lead of a modem, etc.). However, the *close* routine is called only when the last process closes a file, that is, when the i-node table entry is being deallocated. Thus it is not feasible for a device to maintain, or depend on, a count of its users, although it is quite possible to implement an exclusive-use device which cannot be reopened until it has been closed.

When a *read* or *write* takes place, the user's arguments and the *file* table entry are used to set up the variables *u.u_base, u.u_count,* and *u.u_offset* which respectively contain the (user) address of the I/O target area, the byte-count for the transfer, and the current location in the file. If the file referred to is a character-type special file, the appropriate read or write routine is called; it is responsible for transferring data and updating the count and current location appropriately as discussed below. Otherwise, the current location is used to calculate a logical block number in the file. If the file is an ordinary file the logical block number must be mapped (possibly using indirect blocks) to a physical block number; a block-type special file need not be mapped. This mapping is performed by the *bmap* routine. In any event, the resulting physical block number is used, as discussed below, to read or write the appropriate device.

Character Device Drivers

The *cdevsw* table specifies the interface routines present for character devices. Each device provides five routines: open, close, read, write, and special-function (to implement the *ioctl* system call). Any of these may be missing. If a call on the routine should be ignored, (e.g. *open* on non-exclusive devices that require no setup) the *cdevsw* entry can be given as *nulldev;* if it should be considered an error, (e.g. *write* on read-only devices) *nodev* is used. For terminals, the *cdevsw* structure also contains a pointer to the *tty* structure associated with the terminal.

The *open* routine is called each time the file is opened with the full device number as argument. The second argument is a flag which is non-zero only if the device is to be written upon.

The *close* routine is called only when the file is closed for the last time, that is when the very last process in which the file is open closes it. This means it is not possible for the driver to maintain its own count of its users. The first argument is the device number; the second is a flag which is non-zero if the file was open for writing in the process which performs the final *close.*

When *write* is called, it is supplied the device as argument. The per-user variable *u.u_count* has been set to the number of characters indicated by the user; for character devices, this number may be 0 initially. *u.u_base* is the address supplied by the user from which to start taking characters. The system may call the routine internally, so the flag *u.u_segflg* is supplied that indicates, if *on,* that *u.u_base* refers to the system address space instead of the user's.

The *write* routine should copy up to *u.u_count* characters from the user's buffer to the device, decrementing *u.u_count* for each character passed. For most drivers, which work one character at a time, the routine *cpass( )* is used to pick up characters from the user's buffer. Successive calls on it return the characters to be written until *u.u_count* goes to 0 or an error occurs, when it returns −1. *Cpass* takes care of interrogating *u.u_segflg* and updating *u.u_count.*

Write routines which want to transfer a probably large number of characters into an internal buffer may also use the routine *iomove(buffer, offset, count, flag)* which is faster when many characters must be moved. *Iomove* transfers up to *count* characters into the *buffer* starting *offset* bytes from the start of the buffer; *flag* should be *B_WRITE* (which is 0) in the write case. Caution: the caller is responsible for making sure the count is not too large and is non-zero. As an efficiency note, *iomove* is much slower if any of *buffer+offset, count* or *u.u_base* is odd.

The device's *read* routine is called under conditions similar to *write,* except that *u.u_count* is guaranteed to be non-zero. To return characters to the user, the routine *passc(c)* is available; it takes care of housekeeping like *cpass* and returns −1 as the last character specified by *u.u_count* is returned to the user; before that time, 0 is returned. *Iomove* is also usable as with *write;* the flag should be *B_READ* but the same cautions apply.

The ''special-functions'' routine is invoked by the *stty* and *gtty* system calls as follows: *(*p) (dev, v)* where *p* is a pointer to the device's routine, *dev* is the device number, and *v* is a vector. In the *gtty* case, the device is supposed to place up to 3 words of status information into the vector; this will be returned to the caller. In the *stty* case, *v* is 0; the device should take up to 3 words of control information from the array *u.u_arg[0...2].*

Finally, each device should have appropriate interrupt-time routines. When an interrupt occurs, it is turned into a C-compatible call on the devices's interrupt routine. The interrupt-catching mechanism makes the low-order four bits of the ''new PS'' word in the trap vector for the interrupt available to the interrupt handler. This is conventionally used by drivers which deal with multiple similar devices to encode the minor device number. After the interrupt has been processed, a return from the interrupt handler will return from the interrupt itself.

A number of subroutines are available which are useful to character device drivers. Most of these handlers, for example, need a place to buffer characters in the internal interface between their ''top half'' (read/write) and ''bottom half'' (interrupt) routines. For relatively low data-rate devices, the best mechanism is the character queue maintained by the routines *getc* and *putc.* A queue header has the structure

```
struct {
        int     c_cc; /* character count */
        char    *c_cf;/* first character */
        char    *c_cl;/* last character */
} queue;
```

A character is placed on the end of a queue by *putc(c, &queue)* where *c* is the character and *queue* is the queue header. The routine returns −1 if there is no space to put the character, 0 otherwise. The first character on the queue may be retrieved by *getc(&queue)* which returns either the (non-negative) character or −1 if the queue is empty.

Notice that the space for characters in queues is shared among all devices in the system and in the standard system there are only some 600 character slots available. Thus device handlers, especially write routines, must take care to avoid gobbling up excessive numbers of characters.

The other major help available to device handlers is the sleep-wakeup mechanism. The call *sleep(event, priority)* causes the process to wait (allowing other processes to run) until the *event* occurs; at that time, the process is marked ready-to-run and the call will return when there is no process with higher *priority.*

The call *wakeup(event)* indicates that the *event* has happened, that is, causes processes sleeping on the event to be awakened. The *event* is an arbitrary quantity agreed upon by the sleeper and the waker-up. By convention, it is the address of some data area used by the driver, which guarantees that

events are unique.

Processes sleeping on an event should not assume that the event has really happened; they should check that the conditions which caused them to sleep no longer hold.

Priorities can range from 0 to 127; a higher numerical value indicates a less-favored scheduling situation. A distinction is made between processes sleeping at priority less than the parameter *PZERO* and those at numerically larger priorities. The former cannot be interrupted by signals, although it is conceivable that it may be swapped out. Thus it is a bad idea to sleep with priority less than PZERO on an event which might never occur. On the other hand, calls to *sleep* with larger priority may never return if the process is terminated by some signal in the meantime. Incidentally, it is a gross error to call *sleep* in a routine called at interrupt time, since the process which is running is almost certainly not the process which should go to sleep. Likewise, none of the variables in the user area "*u.*" should be touched, let alone changed, by an interrupt routine.

If a device driver wishes to wait for some event for which it is inconvenient or impossible to supply a *wakeup,* (for example, a device going on-line, which does not generally cause an interrupt), the call *sleep(&lbolt, priority)* may be given. *Lbolt* is an external cell whose address is awakened once every 4 seconds by the clock interrupt routine.

The routines *spl4( ), spl5( ), spl6( ), spl7( )* are available to set the processor priority level as indicated to avoid inconvenient interrupts from the device.

If a device needs to know about real-time intervals, then *timeout(func, arg, interval)* will be useful. This routine arranges that after *interval* sixtieths of a second, the *func* will be called with *arg* as argument, in the style *(*func)(arg).* Timeouts are used, for example, to provide real-time delays after function characters like new-line and tab in typewriter output, and to terminate an attempt to read the 201 Dataphone *dp* if there is no response within a specified number of seconds. Notice that the number of sixtieths of a second is limited to 32767, since it must appear to be positive, and that only a bounded number of timeouts can be going on at once. Also, the specified *func* is called at clock-interrupt time, so it should conform to the requirements of interrupt routines in general.

The Block-device Interface

Handling of block devices is mediated by a collection of routines that manage a set of buffers containing the images of blocks of data on the various devices. The most important purpose of these routines is to assure that several processes that access the same block of the same device in multiprogrammed fashion maintain a consistent view of the data in the block. A secondary but still important purpose is to increase the efficiency of the system by keeping in-core copies of blocks that are being accessed frequently. The main data base for this mechanism is the table of buffers *buf.* Each buffer header contains a pair of pointers *(b_forw, b_back)* which maintain a doubly-linked list of the buffers associated with a particular block device, and a pair of pointers *(av_forw, av_back)* which generally maintain a doubly-linked list of blocks which are "free," that is, eligible to be reallocated for another transaction. Buffers that have I/O in progress or are busy for other purposes do not appear in this list. The buffer header also contains the device and block number to which the buffer refers, and a pointer to the actual storage associated with the buffer. There is a word count which is the negative of the number of words to be transferred to or from the buffer; there is also an error byte and a residual word count used to communicate information from an I/O routine to its caller. Finally, there is a flag word with bits indicating the status of the buffer. These flags will be discussed below.

Seven routines constitute the most important part of the interface with the rest of the system. Given a device and block number, both *bread* and *getblk* return a pointer to a buffer header for the block; the difference is that *bread* is guaranteed to return a buffer actually containing the current data for the block, while *getblk* returns a buffer which contains the data in the block only if it is already in core (whether it is or not is indicated by the *B_DONE* bit; see below). In either case the buffer, and the corresponding device block, is made "busy," so that other processes referring to it are obliged to wait until it becomes free. *Getblk* is used, for example, when a block is about to be totally rewritten, so that its previous contents are not useful; still, no other process can be allowed to refer to the block until the new data is placed into it.

The *breada* routine is used to implement read-ahead. it is logically similar to *bread,* but takes as an additional argument the number of a block (on the same device) to be read asynchronously after the specifically requested block is available.

Given a pointer to a buffer, the *brelse* routine makes the buffer again available to other processes. It is called, for example, after data has been extracted following a *bread.* There are three subtly-different write routines, all of which take a buffer pointer as argument, and all of which logically release the buffer for use by others and place it on the free list. *Bwrite* puts the buffer on the appropriate device queue, waits for the write to be done, and sets the user's error flag if required. *Bawrite* places the buffer on the device's queue, but does not wait for completion, so that errors cannot be reflected directly to the user. *Bdwrite* does not start any I/O operation at all, but merely marks the buffer so that if it happens to be grabbed from the free list to contain data from some other block, the data in it will first be written out.

Bwrite is used when one wants to be sure that I/O takes place correctly, and that errors are reflected to the proper user; it is used, for example, when updating i-nodes. *Bawrite* is useful when more overlap is desired (because no wait is required for I/O to finish) but when it is reasonably certain that the write is really required. *Bdwrite* is used when there is doubt that the write is needed at the moment. For example, *bdwrite* is called when the last byte of a *write* system call falls short of the end of a block, on the assumption that another *write* will be given soon which will re-use the same block. On the other hand, as the end of a block is passed, *bawrite* is called, since probably the block will not be accessed again soon and one might as well start the writing process as soon as possible.

In any event, notice that the routines *getblk* and *bread* dedicate the given block exclusively to the use of the caller, and make others wait, while one of *brelse, bwrite, bawrite,* or *bdwrite* must eventually be called to free the block for use by others.

As mentioned, each buffer header contains a flag word which indicates the status of the buffer. Since they provide one important channel for information between the drivers and the block I/O system, it is important to understand these flags. The following names are manifest constants which select the associated flag bits.

B_READ This bit is set when the buffer is handed to the device strategy routine (see below) to indicate a read operation. The symbol *B_WRITE* is defined as 0 and does not define a flag; it is provided as a mnemonic convenience to callers of routines like *swap* which have a separate argument which indicates read or write.

B_DONE This bit is set to 0 when a block is handed to the the device strategy routine and is turned on when the operation completes, whether normally as the result of an error. It is also used as part of the return argument of *getblk* to indicate if 1 that the returned buffer actually contains the data in the requested block.

B_ERROR This bit may be set to 1 when *B_DONE* is set to indicate that an I/O or other error occurred. If it is set the *b_error* byte of the buffer header may contain an error code if it is non-zero. If *b_error* is 0 the nature of the error is not specified. Actually no driver at present sets *b_error;* the latter is provided for a future improvement whereby a more detailed error-reporting scheme may be implemented.

B_BUSY This bit indicates that the buffer header is not on the free list, i.e. is dedicated to someone's exclusive use. The buffer still remains attached to the list of blocks associated with its device, however. When *getblk* (or *bread,* which calls it) searches the buffer list for a given device and finds the requested block with this bit on, it sleeps until the bit clears.

B_PHYS This bit is set for raw I/O transactions that need to allocate the Unibus map on an 11/70.

B_MAP This bit is set on buffers that have the Unibus map allocated, so that the *iodone* routine knows to deallocate the map.

B_WANTEDThis flag is used in conjunction with the *B_BUSY* bit. Before sleeping as described just above, *getblk* sets this flag. Conversely, when the block is freed and the busy bit goes down (in *brelse)* a *wakeup* is given for the block header whenever *B_WANTED* is on. This strategem avoids the overhead of having to call *wakeup* every time a buffer is freed on the

chance that someone might want it.

B_AGE This bit may be set on buffers just before releasing them; if it is on, the buffer is placed at the head of the free list, rather than at the tail. It is a performance heuristic used when the caller judges that the same block will not soon be used again.

B_ASYNC This bit is set by *bawrite* to indicate to the appropriate device driver that the buffer should be released when the write has been finished, usually at interrupt time. The difference between *bwrite* and *bawrite* is that the former starts I/O, waits until it is done, and frees the buffer. The latter merely sets this bit and starts I/O. The bit indicates that *relse* should be called for the buffer on completion.

B_DELWRIThis bit is set by *bdwrite* before releasing the buffer. When *getblk,* while searching for a free block, discovers the bit is 1 in a buffer it would otherwise grab, it causes the block to be written out before reusing it.

Block Device Drivers

The *bdevsw* table contains the names of the interface routines and that of a table for each block device.

Just as for character devices, block device drivers may supply an *open* and a *close* routine called respectively on each open and on the final close of the device. Instead of separate read and write routines, each block device driver has a *strategy* routine which is called with a pointer to a buffer header as argument. As discussed, the buffer header contains a read/write flag, the core address, the block number, a (negative) word count, and the major and minor device number. The role of the strategy routine is to carry out the operation as requested by the information in the buffer header. When the transaction is complete the *B_DONE* (and possibly the *B_ERROR)* bits should be set. Then if the *B_ASYNC* bit is set, *brelse* should be called; otherwise, *wakeup.* In cases where the device is capable, under error-free operation, of transferring fewer words than requested, the device's word-count register should be placed in the residual count slot of the buffer header; otherwise, the residual count should be set to 0. This particular mechanism is really for the benefit of the magtape driver; when reading this device records shorter than requested are quite normal, and the user should be told the actual length of the record.

Although the most usual argument to the strategy routines is a genuine buffer header allocated as discussed above, all that is actually required is that the argument be a pointer to a place containing the appropriate information. For example the *swap* routine, which manages movement of core images to and from the swapping device, uses the strategy routine for this device. Care has to be taken that no extraneous bits get turned on in the flag word.

The device's table specified by *bdevsw* has a byte to contain an active flag and an error count, a pair of links which constitute the head of the chain of buffers for the device *(b_forw, b_back),* and a first and last pointer for a device queue. Of these things, all are used solely by the device driver itself except for the buffer-chain pointers. Typically the flag encodes the state of the device, and is used at a minimum to indicate that the device is currently engaged in transferring information and no new command should be issued. The error count is useful for counting retries when errors occur. The device queue is used to remember stacked requests; in the simplest case it may be maintained as a first-in first-out list. Since buffers which have been handed over to the strategy routines are never on the list of free buffers, the pointers in the buffer which maintain the free list *(av_forw, av_back)* are also used to contain the pointers which maintain the device queues.

A couple of routines are provided which are useful to block device drivers. *iodone(bp)* arranges that the buffer to which *bp* points be released or awakened, as appropriate, when the strategy module has finished with the buffer, either normally or after an error. (In the latter case the *B_ERROR* bit has presumably been set.)

The routine *geterror(bp)* can be used to examine the error bit in a buffer header and arrange that any error indication found therein is reflected to the user. It may be called only in the non-interrupt part of a driver when I/O has completed *(B_DONE* has been set).

Raw Block-device I/O

A scheme has been set up whereby block device drivers may provide the ability to transfer information directly between the user's core image and the device without the use of buffers and in blocks as large as the caller requests. The method involves setting up a character-type special file corresponding to the raw device and providing *read* and *write* routines which set up what is usually a private, non-shared buffer header with the appropriate information and call the device's strategy routine. If desired, separate *open* and *close* routines may be provided but this is usually unnecessary. A special-function routine might come in handy, especially for magtape.

A great deal of work has to be done to generate the ''appropriate information'' to put in the argument buffer for the strategy module; the worst part is to map relocated user addresses to physical addresses. Most of this work is done by *physio(strat, bp, dev, rw)* whose arguments are the name of the strategy routine *strat,* the buffer pointer *bp,* the device number *dev,* and a read-write flag *rw* whose value is either *B_READ* or *B_WRITE. Physio* makes sure that the user's base address and count are even (because most devices work in words) and that the core area affected is contiguous in physical space; it delays until the buffer is not busy, and makes it busy while the operation is in progress; and it sets up user error return information.

UNIX Programming — Second Edition

Brian W. Kernighan

Dennis M. Ritchie

AT&T Bell Laboratories
Murray Hill, NJ 07974

ABSTRACT

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

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

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

1. INTRODUCTION

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

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

2. BASICS

2.1. Program Arguments

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

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

---

† UNIX is a registered trademark of The Open Group in the U.S. and other countries.

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

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

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

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

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

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

```
prog <file
```

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

```
otherprog | prog
```

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

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

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

```
prog >outfile
```

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

```
prog | otherprog
```

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

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

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

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

```
#include <stdio.h>

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

The line

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

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

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

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

and thus avoid learning how to access files from a program. By the way, the call to `exit` at the end is not necessary to make the program work properly, but it assures that any caller of the program will see a normal termination status (conventionally 0) from the program when it completes. Section 6 discusses status returns in more detail.

3. THE STANDARD I/O LIBRARY

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

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

3.1. File Access

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

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

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

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

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

This internal name is actually a pointer, called a *file pointer*, to a structure which contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and the like. Users don't need to know the details, because

part of the standard I/O definitions obtained by including `stdio.h` is a structure definition called `FILE`. The only declaration needed for a file pointer is exemplified by

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

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

The actual call to `fopen` in a program is

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

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

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

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

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

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

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

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

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

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

```
#include <stdio.h>

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

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

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

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

3.2. Error Handling — Stderr and Exit

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

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

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

3.3. Miscellaneous I/O Functions

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

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

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

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

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

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

4. LOW-LEVEL I/O

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

4.1. File Descriptors

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

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

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

Since input and output involving the user's terminal are so common, special arrangements exist to make this convenient. When the command interpreter (the "shell") runs a program, it opens three files, with file descriptors 0, 1, and 2, called the standard input, the standard output, and the standard error output. All of these are normally connected to the terminal, so if a program reads file descriptor 0 and

writes file descriptors 1 and 2, it can do terminal I/O without worrying about opening the files.

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

```
prog <infile >outfile
```

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

4.2. Read and Write

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.3. Open, Creat, Close, Unlink

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

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

```
int fd;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.4. Random Access — Seek and Lseek

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

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

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

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

To get back to the beginning (“rewind”),

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

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

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

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

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

4.5. Error Processing

The routines discussed in this section, and in fact all the routines which are direct entries into the system can incur errors. Usually they indicate an error by returning a value of –1. Sometimes it is nice to know what sort of error occurred; for this purpose all these routines, when appropriate, leave an error number in the external cell `errno`. The meanings of the various error numbers are listed in the introduction to Section II of the *UNIX Programmer's Manual,* so your program can, for example, determine if an attempt to open a file failed because it did not exist or because the user lacked permission to read it. Perhaps more commonly, you may want to print out the reason for failure. The routine `perror` will print a message associated with the value of `errno`; more generally, `sys_errno` is an array of character strings which can be indexed by `errno` and printed by your program.

5. PROCESSES

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

5.1. The "System" Function

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

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

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

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

5.2. Low-Level Process Creation — Execl and Execv

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.3. Control of Processes — Fork and Wait

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

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

splits the program into two copies, both of which continue to run. The only difference between the two is the value of `proc_id`, the "process id." In one of these processes (the "child"), `proc_id` is zero.

In the other (the ‘‘parent’’), `proc_id` is non-zero; it is the process number of the child. Thus the basic way to call, and return from, another program is

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

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

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

```
int status;

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

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

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

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

5.4. Pipes

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

```
ls | pr
```

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

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

```
int  fd[2];

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

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

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

closed, a subsequent `read` will encounter end of file.

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

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

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

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

```
#include <stdio.h>

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

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

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

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

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

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

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

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

```
#include <signal.h>

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

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

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

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

6. SIGNALS — INTERRUPTS AND ALL THAT

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

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

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

causes interrupts to be ignored, while

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

restores the default action of process termination. In all cases, `signal` returns the previous value of the signal. The second argument to `signal` may instead be the name of a function (which has to be declared explicitly if the compiler hasn't seen it already). In this case, the named routine will be called when the signal occurs. Most commonly this facility is used to allow the program to clean up unfinished business before terminating, for example to delete a temporary file:

```
#include <signal.h>

main()
{
     int onintr();

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

     /* Process ... */

     exit(0);
}

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

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

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

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

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

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

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

     /* main processing loop */
}

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

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

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

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

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

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

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

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

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

```
#include <signal.h>

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

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

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

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

References

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

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

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

Appendix — The Standard I/O Library

D. M. Ritchie

AT&T Bell Laboratories
Murray Hill, NJ 07974

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

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

1. General Usage

Each program using the library must have the line

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

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

stdin The name of the standard input file

stdout The name of the standard output file

stderr The name of the standard error file

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

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

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

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

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

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

2. Calls

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`ungetc(c, ioptr) FILE *ioptr;`
The argument character `c` is pushed back on the input stream named by `ioptr`. Only one character may be pushed back.

`printf(format, a1, ...) char *format;`
`fprintf(ioptr, format, a1, ...) FILE *ioptr; char *format;`
`sprintf(s, format, a1, ...)char *s, *format;`
`printf` writes on the standard output. `fprintf` writes on the named output stream. `sprintf` puts characters in the character array (string) named by `s`. The specifications are as described in section `printf`(3) of the *UNIX Programmer's Manual.*

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

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

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

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

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

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

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

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

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

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

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

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

`fseek(ioptr, offset, ptrname) FILE *ioptr; long offset;`
The location of the next byte in the stream named by `ioptr` is adjusted. `offset` is a long integer. If `ptrname` is 0, the offset is measured from the beginning of the file; if `ptrname` is 1, the offset is measured from the current read or write pointer; if `ptrname` is 2, the offset is measured from the end of the file. The routine accounts properly for any buffering. (When this routine is used on non-UNIX systems, the offset must be a value returned from `ftell` and the ptrname must be 0).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Berkeley Software Architecture Manual
4.4BSD Edition

M. Kirk McKusick, Michael Karels

Samuel Leffler, William Joy

Robert Fabry

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

ABSTRACT

This document summarizes the system calls provided by the 4.4BSD 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 process: process naming and protection, memory management, software interrupts, time and statistics functions, object references (descriptors), 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 filesystems, communication, terminal handling, and process control and debugging. These facilities are implemented by the operating system or by network server processes.

Notation and Types

The notation used to describe system calls is a variant of a C language function call, consisting of a prototype call followed by the 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 2.1.1:

```
cc = read(fd, buf, nbytes);
result ssize_t cc; int fd; result void *buf; size_t nbytes;
```

The first line shows how the *read* routine is called, with three parameters. As shown on the second line, the return value *cc* is a size_t and *read* also returns information in the parameter *buf*.

The descriptions of error conditions arising from each system call are not provided here; they appear in section 2 of the Programmer's Reference 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 by the include file *<sys/types.h>* and used in the specifications here and in many C programs.

```
Type        Value
------------------------------------------------------
caddr_t     char *              /* a memory address */
clock_t     unsigned long       /* count of CLK_TCK's */
gid_t       unsigned long       /* group ID */
int16_t     short               /* 16-bit integer */
int32_t     int                 /* 32-bit integer */
int64_t     long long           /* 64-bit integer */
int8_t      signed char         /* 8-bit integer */
mode_t      unsigned short      /* file permissions */
off_t       quad_t              /* file offset */
pid_t       long                /* process ID */
qaddr_t     quad_t *
quad_t      long long
size_t      unsigned int        /* count of bytes */
ssize_t     int                 /* signed size_t */
time_t      long                /* seconds since the Epoch */
u_char      unsigned char
u_int       unsigned int
u_int16_t   unsigned short      /* unsigned 16-bit integer */
u_int32_t   unsigned int        /* unsigned 32-bit integer */
u_int64_t   unsigned long long  /* unsigned 64-bit integer */
u_int8_t    unsigned char       /* unsigned 8-bit integer */
u_long      unsigned long
u_quad_t    unsigned long long
u_short     unsigned short
uid_t       unsigned long       /* user ID */
uint        unsigned int        /* System V compatibility */
ushort      unsigned short      /* System V compatibility */
```

1. Kernel primitives

The facilities available to a user process are logically divided into two parts: kernel facilities directly implemented by code running in the operating system, and system facilities implemented either by the system, or in cooperation with a *server process*. The kernel facilities are described in section 1.

The facilities implemented in the kernel are those which define the *4.4BSD virtual machine* in which each process runs. Like many real machines, this virtual machine has memory management hardware, an interrupt facility, timers and counters. The 4.4BSD virtual machine 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 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 identifiers

Each host has associated with it an integer host ID, and a host name of up to MAXHOSTNAMELEN (256) characters (as defined in <*sys/param.h*>). These identifiers are set (by a privileged user) and retrieved using the *sysctl* interface described in section 1.7.1. The host ID is seldom used (or set), and is deprecated. For convenience and backward compatibility, the following library routines are provided:

```
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;
```

1.1.2. Process identifiers

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 pid_t pid;
```

On each host this identifier is guaranteed to be unique; in a multi-host environment, the (hostid, process ID) pairs are guaranteed unique. The parent process identifier can be obtained using the *getppid* routine:

```
pid = getppid();
result pid_t pid;
```

1.1.3. Process creation and termination

A new process is created by making a logical duplicate of an existing process:

```
pid = fork();
result pid_t 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 imposes a hierarchical

structure on the set of processes in the system.

For processes that are forking solely for the purpose of *execve*'ing another program, the *vfork* system call provides a faster interface:

```
pid = vfork();
result pid_t pid;
```

Like *fork*, the *vfork* 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 process is suspended until the child process calls either *execve* or *exit*.

A process may terminate by executing an *exit* call:

```
exit(status);
int status;
```

The lower 8 bits of exit status are available to its parent.

When a child process exits or terminates abnormally, the parent process receives information about the event which caused termination of the child process. The interface allows the parent to wait for a particular process, process group, or any direct descendent and to retrieve information about resources consumed by the process during its lifetime. The request may be done either synchronously (waiting for one of the requested processes to exit), or asynchronously (polling to see if any of the requested processes have exited):

```
pid = wait4(wpid, astatus, options, arusage);
result pid_t pid; pid_t wpid; result int *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. If the set-user-ID mode bit is set, the effective user ID is set to the owner of the file; if the set-group-ID mode bit is set, the effective group ID is set to the group of the file. Whether changed or not, the effective user ID is then copied to the saved user ID, and the effective group ID is copied to the saved group ID.

1.1.4. User and group IDs

Each process in the system has associated with it three user IDs: a *real user ID*, an *effective user ID*, and a *saved user ID*, all unsigned integral types (**uid_t**). Each process has a *real group ID* and a set of *access group IDs*, the first of which is the *effective group ID*. The group IDs are unsigned integral types (**gid_t**). Each process may be in multiple access groups. The maximum concurrent number of access groups is a system compilation parameter, represented by the constant NGROUPS in the file *<sys/param.h>*. It is guaranteed to be at least 16.

The real group ID is used in process accounting and in testing whether the effective group ID may be changed; it is not otherwise used for access control. The members of the access group ID set are used for access control. Because the first member of the set is the effective group ID, which is changed when executing a set-group-ID program, that element is normally duplicated in the set so that access privileges for the original group are not lost when using a set-group-ID program.

The real and effective user IDs associated with a process are returned by:

```
ruid = getuid();
result uid_t ruid;
```

```
euid = geteuid();
result uid_t euid;
```

the real and effective group IDs by:

```
rgid = getgid();
result gid_t rgid;

egid = getegid();
result gid_t egid;
```

The access group ID set is returned by a *getgroups* call:

```
ngroups = getgroups(gidsetsize, gidset);
result int ngroups; int gidsetsize; result gid_t gidset[gidsetsize];
```

The user and group IDs are assigned at login time using the *setuid*, *setgid*, and *setgroups* calls:

```
setuid(uid);
uid_t uid;

setgid(gid);
gid_t gid;

setgroups(gidsetsize, gidset);
int gidsetsize; gid_t gidset[gidsetsize];
```

The *setuid* call sets the real, effective, and saved user IDs, and is permitted only if the specified *uid* is the current real user ID or if the caller is the super-user. The *setgid* call sets the real, effective, and saved group IDs; it is permitted only if the specified *gid* is the current real group ID or if the caller is the super-user. The *setgroups* call sets the access group ID set, and is restricted to the super-user.

The *seteuid* routine allows any process to set its effective user ID to either its real or saved user ID:

```
seteuid(uid);
uid_t uid;
```

The *setegid* routine allows any process to set its effective group ID to either its real or saved group ID:

```
setegid(gid);
gid_t gid;
```

1.1.5. Sessions

When a user first logs onto the system, they are put into a session with a controlling process (usually a shell). The session is created with the call:

```
pid = setsid();
result pid_t pid;
```

All subsequent processes created by the user (that do not call *setsid*) will be part of the session. The session also has a login name associated with it which is set using the privileged call:

```
setlogin(name);
char *name;
```

The login name can be retrieved using the call:

```
name = getlogin();
result char *name;
```

Unlike historic systems, the value returned by *getlogin* is stored in the kernel and can be trusted.

1.1.6. Process groups

Each process in the system is also 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). All members of a process group are members of the same session. The current process group of a process is returned by the *getpgrp* call:

```
pgrp = getpgrp();
result pid_t pgrp;
```

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 a terminal among several different jobs.

The process group associated with a process may be changed by the *setpgid* call:

```
setpgid(pid, pgrp);
pid_t pid, pgrp;
```

Newly created processes are assigned process IDs distinct from all processes and process groups, and the same process group as their parent. Any process may set its process group equal to its process ID or to the value of any process group within its session.

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 writable and private to the process. Both the data and stack areas may be extended and contracted on program request. The call:

```
brk(addr);
caddr_t addr;
```

sets the end of the data segment to the specified address. More conveniently, the end can be extended by *incr* bytes, and the base of the new area returned with the call:

```
addr = sbrk(incr);
result caddr_t addr; int incr;
```

Application programs normally use the library routines *malloc* and *free*, which provide a more convenient interface than *brk* and *sbrk*.

There is no call for extending the stack, as it is automatically extended as needed.

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 <*sys/mman.h*> as:

Protections are chosen from these bits, or-ed together:

```
PROT_READ       /* pages can be read */
PROT_WRITE      /* pages can be written */
PROT_EXEC       /* pages can be executed */
```

Flags contain sharing type and options. Sharing options, choose one:

```
MAP_SHARED                  /* share changes */
MAP_PRIVATE                 /* changes are private */
```

Option flags†:

```
MAP_ANON            /* allocated from virtual memory; fd ignored */
MAP_FIXED           /* map addr must be exactly as requested */
MAP_NORESERVE       /* don't reserve needed swap area */
MAP_INHERIT         /* region is retained after exec */
MAP_HASSEMAPHORE    /* region may contain semaphores */
```

The size of a page is cpu-dependent, and is returned by the *sysctl* interface described in section 1.7.1. The *getpagesize* library routine is provided for convenience and backward compatibility:

```
pagesize = getpagesize();
result int pagesize;
```

The call:

```
maddr = mmap(addr, len, prot, flags, fd, pos);
result caddr_t maddr; caddr_t addr; size_t len; int prot, flags, fd; off_t pos;
```

causes the pages starting at *addr* and continuing for at most *len* bytes to be mapped from the object represented by descriptor *fd*, starting at byte offset *pos*. If *addr* is NULL, the system picks an unused address for the region. The starting address of the region is returned; for the convenience of the system, it may differ from that supplied unless the MAP_FIXED flag is given, in which case the exact address will be used or the call will fail. The *addr* parameter must be a multiple of the pagesize (if MAP_FIXED is given). If *pos* and *len* are not a multiple of pagesize, they will be rounded (down and up respectively) to a page boundary by the system; the rounding will cause the mapped region to extend past the specified range. A successful *mmap* will delete any previous mapping in the allocated address range. The parameter *prot* specifies the accessibility of the mapped pages. The parameter *flags* specifies the type of object to be mapped, mapping options, and whether modifications made to this mapped copy of the page are to be kept *private*, or are to be *shared* with other references. Possible types include MAP_SHARED or MAP_PRIVATE that map a regular file or character-special device memory, and MAP_ANON, which maps memory not associated with any specific file. The file descriptor used when creating MAP_ANON regions is not used and should be –1. The MAP_INHERIT flag allows a region to be inherited after an *execve*. The MAP_HASSEMAPHORE flag allows special handling for regions that may contain semaphores. The MAP_NORESERVE flag allows processes to allocate regions whose virtual address space, if fully allocated, would exceed the available memory plus swap resources. Such regions may get a SIGSEGV signal if they page fault and resources are not available to service their request; typically they would free up some resources via *munmap* so that when they return from the signal the page fault could be completed successfully.

A facility is provided to synchronize a mapped region with the file it maps; the call:

```
msync(addr, len);
caddr_t addr; size_t len;
```

causes any modified pages in the specified region to be synchronized with their source and other mappings. If necessary, it writes any modified pages back to the filesystem, and updates the file modification time. If *len* is 0, all modified pages within the region containing *addr* will be flushed; this usage is

† In 4.4BSD, only MAP_ANON and MAP_FIXED are implemented.

provisional, and may be withdrawn. If *len* is non-zero, only the pages containing *addr* and *len* succeeding locations will be examined. Any required synchronization of memory caches will also take place at this time.

Filesystem operations on a file that is mapped for shared modifications are currently unpredictable except after an *msync*.

A mapping can be removed by the call

```
munmap(addr, len);
caddr_t addr; size_t len;
```

This call deletes the mappings for the specified address range, and causes further references to addresses within the range to generate invalid memory references.

1.2.3. Page protection control

A process can control the protection of pages using the call:

```
mprotect(addr, len, prot);
caddr_t addr; size_t len; int prot;
```

This call changes the specified pages to have protection *prot*. Not all implementations will guarantee protection on a page basis; the granularity of protection changes may be as large as an entire region.

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; size_t len; int behav;
```

Behav describes expected behavior, as given in <*sys/mman.h*>:

```
MADV_NORMAL      /* no further special treatment */
MADV_RANDOM      /* expect random page references */
MADV_SEQUENTIAL  /* expect sequential references */
MADV_WILLNEED    /* will need these pages */
MADV_DONTNEED    /* don't need these pages */
```

The *mincore*† function allows a process to obtain information about whether pages are memory resident:

```
mincore(addr, len, vec);
caddr_t addr; size_t len; result char *vec;
```

Here the current memory residency of the pages is returned in the character array *vec*, with a value of 1 meaning that the page is in-memory. *Mincore* provides only transient information about page residency. Real-time processes that need guaranteed residence over time can use the call:

```
mlock(addr, len);
caddr_t addr; size_t len;
```

This call locks the pages for the specified address range into memory (paging them in if necessary) ensuring that further references to addresses within the range will never generate page faults. The amount of memory that may be locked is controlled by a resource limit, see section 1.6.3. When the memory is no longer critical it can be unlocked using:

---

† The entry point for this system call is defined, but is not implemented, so currently always returns with the error "Operation not supported."

```
munlock(addr, len);
caddr_t addr; size_t len;
```

After the *munlock* call, the pages in the specified address range are still accessible but may be paged out if memory is needed and they are not accessed.

1.2.5. Synchronization primitives

Primitives are provided for synchronization using semaphores in shared memory.‡ These primitives are expected to be superseded by the semaphore interface being specified by the POSIX 1003 Pthread standard. They are provided as an efficient interim solution. Application programmers are encouraged to use the Pthread interface when it becomes available.

Semaphores must lie within a MAP_SHARED region with at least modes PROT_READ and PROT_WRITE. The MAP_HASSEMAPHORE flag must have been specified when the region was created. To acquire a lock a process calls:

```
value = mset(sem, wait);
result int value; semaphore *sem; int wait;
```

Mset indivisibly tests and sets the semaphore *sem*. If the previous value is zero, the process has acquired the lock and *mset* returns true immediately. Otherwise, if the *wait* flag is zero, failure is returned. If *wait* is true and the previous value is non-zero, *mset* relinquishes the processor until notified that it should retry.

To release a lock a process calls:

```
mclear(sem);
semaphore *sem;
```

Mclear indivisibly tests and clears the semaphore *sem*. If the ''WANT'' flag is zero in the previous value, *mclear* returns immediately. If the ''WANT'' flag is non-zero in the previous value, *mclear* arranges for waiting processes to retry before returning.

Two routines provide services analogous to the kernel *sleep* and *wakeup* functions interpreted in the domain of shared memory. A process may relinquish the processor by calling *msleep* with a set semaphore:

```
msleep(sem);
semaphore *sem;
```

If the semaphore is still set when it is checked by the kernel, the process will be put in a sleeping state until some other process issues an *mwakeup* for the same semaphore within the region using the call:

```
mwakeup(sem);
semaphore *sem;
```

An *mwakeup* may awaken all sleepers on the semaphore, or may awaken only the next sleeper on a queue.

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 a *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

‡ All currently unimplemented, no entry points exists.

default action is to be taken when signals occur.

Some signals will cause a process to exit if 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 also 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, signals that may be generated by the program's action are delivered first; the order in which other signals are delivered to a process is not specified. Signal routines execute with the signal that caused their invocation *blocked*, but other signals may occur. Multiple signals may be delivered on a single entry to the system, as if signal handlers were interrupted by other signal handlers. 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 by 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 attempts to access addresses outside the currently assigned area of memory, and SIGBUS for accesses that violate memory access constraints.

Software signals reflect interrupts generated by user request: SIGINT for the normal interrupt signal; SIGQUIT for the more powerful *quit* signal, which 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. Programs may define their own asynchronous events using SIGUSR1 and SIGUSR2. Other software signals (SIGALRM, SIGVTALRM, SIGPROF) indicate the expiration of interval timers. When a window changes size, a SIGWINCH is sent to the controlling terminal process group.

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 for stopping the process. 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 when a process reaches the limit on file size.

1.3.3. Signal handlers

A process has a handler associated with each signal. The handler controls the way the signal is delivered. The call:

```
struct sigaction {
        void        (*sa_handler)();
        sigset_t    sa_mask;
        int         sa_flags;
};
```

```
sigaction(signo, sa, osa);
int signo; struct sigaction *sa; result struct sigaction *osa;
```

assigns interrupt handler address *sa_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_DFL used as values for *sa_handler* cause ignoring or defaulting of a condition, respectively. The *sa_mask* value specifies the signal mask to be used when the handler is invoked; it implicitly includes the signal which invoked the handler. Signal masks include one bit for each signal. The following macros, defined in *signal.h*, create an empty mask, then put *signo* into it:

```
sigemptyset(set);
sigaddset(set, signo);
result sigset_t *set; int signo;
```

Sa_flags specifies whether pending system calls should be restarted if the signal handler returns (SA_RESTART) and whether the handler should operate on the normal run-time stack or a special signal stack (SA_ONSTACK; see below). If *osa* is non-zero, the previous signal handler information 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 it then will be delivered. The process of signal delivery adds the signal to be delivered and those signals specified in the associated signal handler's *sa_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 returns normally, the signal mask will be restored and the process will resume execution in the original context.

The mask of *blocked* signals is independent of handlers for signals. It delays signals from being delivered much as a raised hardware interrupt priority level delays hardware interrupts. Preventing an interrupt from occurring by changing the handler is analogous to disabling a device from further interrupts.

The signal handling routine *sa_handler* is called by a C call of the form:

```
(*sa_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 signal-specific 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. Normally this context will be restored when the signal handler returns. However, a process may do so at any time by using the call:

```
sigreturn(scp);
struct sigcontext *scp;
```

If the signal handler makes a call to *longjmp*, the signal mask at the time of the corresponding *setjmp* is restored.

1.3.4. Sending signals

A process can send a signal to another process or processes group with the call:

```
kill(pid, signo)
pid_t pid; int signo;
```

For compatibility with old systems, a compatibility routine is provided to send a signal to a process group:

```
killpg(pgrp, signo)
pid_t pgrp; int signo;
```

Unless the process sending the signal is privileged, it must have the same effective user id as the process receiving the signal.

Signals also are 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

The *sigprocmask* system call is used to manipulate the mask of blocked signals:

```
sigprocmask(how, newmask, oldmask);
int how; sigset_t *newmask; result sigset_t *oldmask;
```

The actions done by *sigprocmask* are to add to the list of masked signals (SIG_BLOCK), delete from the list of masked signals (SIG_UNBLOCK), and block a specific set of signals (SIG_SETMASK). The *sigprocmask* call can be used to read the current mask by specifying SIG_BLOCK with an empty *newmask* .

It is possible to check conditions with some signals blocked, and then to pause waiting for a signal and restoring the mask, by using:

```
sigsuspend(mask);
sigset_t *mask;
```

It is also possible to find out which blocked signals are pending delivery using the call:

```
sigpending(mask);
result sigset_t *mask;
```

1.3.6. Signal stacks

Applications that maintain complex or fixed size stacks can use the call:

```
struct sigaltstack {
        caddr_t     ss_sp;
        long        ss_size;
        int         ss_flags;
};


sigaltstack(ss, oss)
struct sigaltstack *ss; result struct sigaltstack *oss;
```

to provide the system with a stack based at *ss_sp* of size *ss_size* for delivery of signals. The value *ss_flags* 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 to a handler for which the SA_ONSTACK flag was set, 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 doing a *sigreturn* to restore the previous stack. If the process takes a non-local exit from the signal routine, *longjmp* will do a *sigreturn* call to switch back to the run-time stack.

1.4. Timers

1.4.1. Real time

The system's notion of the current time is in Coordinated Universal Time (UTC, previously GMT) and the current time zone is set and returned by the calls:

```
settimeofday(tp, 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 */
};
```

The timezone information is present only for historical reasons and is unused by the current system.

The precision of the system clock is hardware dependent. Earlier versions of UNIX contained only a 1-second resolution version of this call, which remains as a library routine:

```
time(tvsec);
result time_t *tvsec;
```

returning only the tv_sec field from the *gettimeofday* call.

The *adjtime* system calls allows for small changes in time without abrupt changes by skewing the rate at which time advances:

```
adjtime(delta, olddelta);
struct timeval *delta; result struct timeval *olddelta;
```

1.4.2. Interval time

The system provides each process with three interval timers, defined in *<sys/time.h>*:

```
ITIMER_REAL     /* real time intervals */
ITIMER_VIRTUAL  /* virtual time intervals */
ITIMER_PROF     /* 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:

```
setitimer(which, value, ovalue);
int which; struct itimerval *value; result struct itimerval *ovalue;

getitimer(which, value);
int which; result struct itimerval *value;
```

The *it_value* specifies the time until the next signal; the *it_interval* specifies a new interval that should be loaded into the timer on each expiration. The third argument to *setitimer* specifies an optional structure to receive the previous contents of the interval timer. A timer can be disabled by setting *it_value* and *it_interval* to 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. The *profil* call arranges for the kernel to begin gathering execution statistics for a process:

```
profil(samples, size, offset, scale);
result char *samples; int size, offset, scale;
```

This call begins sampling the program counter, with statistics maintained in the user-provided buffer.

1.5. Descriptors

1.5.1. The reference table

Each process has access to resources through *descriptors*. Each descriptor is a handle allowing processes 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 therefore may have multiple references, and are reference counted by the system.

Each process has a limited size descriptor reference table, where the current size is returned by the *getdtablesize* call:

```
nds = getdtablesize();
result int nds;
```

and guaranteed to be at least 64. The maximum number of descriptors is a resource limit (see section 1.6.3). The entries in the descriptor reference table are referred to by small integers; for example if there are 64 slots they are numbered 0 to 63.

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. For those types that support random access, the current file offset is stored in the descriptor. 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* which is indistinguishable from the original. The value of *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.

Descriptors are deallocated 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 to examine the state of multiple descriptors simultaneously, and to wait for state changes on those descriptors. Sets of descriptors of interest are specified as bit masks, as follows:

```
nds = select(nd, in, out, except, tvp);
result int nds; int nd; result fd_set *in, *out, *except;
struct timeval *tvp;

FD_CLR(fd, &fdset);
FD_COPY(&fdset, &fdset2);
FD_ISSET(fd, &fdset);
FD_SET(fd, &fdset);
FD_ZERO(&fdset);
int fd; fs_set fdset, fdset2;
```

The *select* call examines the descriptors specified by the sets *in*, *out* and *except*, replacing the specified bit masks by the subsets that select true for input, output, and exceptional conditions respectively (*nd* indicates the number of file descriptors specified by 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 sections 2.1.3 and 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 sections 2.1.3 and 2.3.1.5).
- 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), or other device-specific events have occurred.

For these tests, an operation is considered to be possible if a call to the operation would return without blocking (even if the O_NONBLOCK flag were not set). For example, a descriptor would test as ready for reading if a read call would return immediately with data, an end-of-file indication, or an error other than EWOULDBLOCK.

If none of the specified conditions is true, the operation waits for one of the conditions to arise, blocking at most the amount of time specified by *tvp*. If *tvp* is given as NULL, the *select* waits indefinitely.

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;


/* command values */

F_DUPFD     /* return a new descriptor */
F_GETFD     /* get file descriptor flags */
F_SETFD     /* set file descriptor flags */
F_GETFL     /* get file status flags */
F_SETFL     /* set file status flags */
F_GETOWN    /* get SIGIO/SIGURG proc/pgrp */
F_SETOWN    /* set SIGIO/SIGURG proc/pgrp */
F_GETLK     /* get blocking lock */
F_SETLK     /* set or clear lock */
F_SETLKW    /* set lock with wait */
```

The F_DUPFD *cmd* provides identical functionality to *dup2*; it is provided solely for POSIX compatibility. The F_SETFD *cmd* can be used to set the close-on-exec flag for a file descriptor. The F_SETFL *cmd* may be used to set a descriptor in non-blocking I/O mode and/or enable signaling when I/O is possible. F_SETOWN may be used to specify a process or process group to be signaled when using the latter mode of operation or when urgent indications arise. The *fcntl* system call also provides POSIX-compliant byte-range locking on files. However the semantics of unlocking on every *close* rather than last close makes them useless. Much better semantics and faster locking are provided by the *flock* system call (see section 2.2.7). The *fcntl* and *flock* locks can be used concurrently; they will serialize against each other properly.

Operations on non-blocking descriptors will either complete immediately, return the 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, then 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.

1.6. 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. The instantaneous scheduling priority is a function of CPU usage and a settable priority value used in adjusting the instantaneous priority with CPU usage or inactivity. It is possible to determine the settable priority factor currently assigned to a process (PRIO_PROCESS), process group (PRIO_PGRP), or the processes of

a specified user (PRIO_USER), or to alter this priority using the calls:

```
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 the specified processes to the specified value. Only the super-user may lower priorities.

1.6.2. Resource utilization

The *getrusage* call returns information describing the resources utilized by the current process (RUSAGE_SELF), or all its terminated descendent processes (RUSAGE_CHILDREN):

```
getrusage(who, rusage);
int who; result struct rusage *rusage;
```

The information is returned in a structure defined in <*sys/resource.h*>:

```
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 memory size */
        int     ru_isrss;               /* unshared stack memory size */
        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 operations */
        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 context switches */
};
```

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:

```
getrlimit(resource, rlp);
int resource; result struct rlimit *rlp;

setrlimit(resource, rlp);
int resource; struct rlimit *rlp;
```

The resources that may currently be controlled include:

```
RLIMIT_CPU      /* cpu time in milliseconds */
RLIMIT_FSIZE    /* maximum file size */
RLIMIT_DATA     /* data size */
RLIMIT_STACK    /* stack size */
RLIMIT_CORE     /* core file size */
RLIMIT_RSS      /* resident set size */
RLIMIT_MEMLOCK  /* locked-in-memory address space */
RLIMIT_NPROC    /* number of processes */
RLIMIT_NOFILE   /* number of open files */
```

Each limit has a current value and a maximum defined by the *rlimit* structure:

```
struct rlimit {
        quad_t  rlim_cur;       /* current (soft) limit */
        quad_t  rlim_max;       /* hard limit */
};
```

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*. To remove a limit on a resource, the value is set to RLIM_INFINITY.

1.7. System operation support

Unless noted otherwise, the calls in this section are permitted only to a privileged user.

1.7.1. Monitoring system operation

The *sysctl* function allows any process to retrieve system information and allows processes with appropriate privileges to set system configurations.

```
sysctl(name, namelen, oldp, oldlenp, newp, newlen);
int *name; u_int namelen; result void *oldp; result size_t *oldlenp;
void *newp; size_t newlen;
```

The information available from *sysctl* consists of integers, strings, and tables. *Sysctl* returns a consistent snapshot of the data requested. Consistency is obtained by locking the destination buffer into memory so that the data may be copied out without blocking. Calls to *sysctl* are serialized to avoid deadlock.

The object to be interrogated or set is named using a ''Management Information Base'' (MIB) style name, listed in *name*, which is a *namelen* length array of integers. This name is from a hierarchical name space, with the most significant component in the first element of the array. It is analogous to a file pathname, but with integers as components rather than slash-separated strings.

The information is copied into the buffer specified by *oldp*. The size of the buffer is given by the location specified by *oldlenp* before the call, and that location is filled in with the amount of data copied after a successful call. If the amount of data available is greater than the size of the buffer supplied, the call supplies as much data as fits in the buffer provided and returns an error.

To set a new value, *newp* is set to point to a buffer of length *newlen* from which the requested value is to be taken. If a new value is not to be set, *newp* should be set to NULL and *newlen* set to 0.

The top level names (those used in the first element of the *name* array) are defined with a CTL_ prefix in *<sys/sysctl.h>*, and are as follows. The next and subsequent levels down are found in the include files listed here:

Name	Next Level Names	Description
CTL_DEBUG	sys/sysctl.h	Debugging
CTL_FS	sys/sysctl.h	Filesystem
CTL_HW	sys/sysctl.h	Generic CPU, I/O
CTL_KERN	sys/sysctl.h	High kernel limits
CTL_MACHDEP	sys/sysctl.h	Machine dependent
CTL_NET	sys/socket.h	Networking
CTL_USER	sys/sysctl.h	User-level
CTL_VM	vm/vm_param.h	Virtual memory

1.7.2. Bootstrap operations

The call:

```
mount(type, dir, flags, data);
int type; char *dir; int flags; caddr_t data;
```

extends the name space. The *mount* call grafts a filesystem object onto the system file tree at the point specified in *dir*. The argument *type* specifies the type of filesystem to be mounted. The argument *data* describes the filesystem object to be mounted according to the *type*. The contents of the filesystem become available through the new mount point *dir*. Any files in or below *dir* at the time of a successful mount disappear from the name space until the filesystem is unmounted. The *flags* value specifies generic properties, such as a request to mount the filesystem read-only.

Information about all mounted filesystems can be obtained with the call:

```
getfsstat(buf, bufsize, flags);
result struct statfs *buf; long bufsize, int flags;
```

The call:

```
swapon(blkdev);
char *blkdev;
```

specifies a device to be made available for paging and swapping.

1.7.3. Shutdown operations

The call:

```
unmount(dir, flags);
char *dir; int flags;
```

unmounts the filesystem mounted on *dir*. This call will succeed only if the filesystem is not currently being used or if the MNT_FORCE flag is specified.

The call:

```
sync();
```

schedules I/O to flush all modified disk blocks resident in the kernel. (This call does not require privileged status.) Files can be selectively flushed to disk using the *fsync* call (see section 2.2.6).

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, among other options. These constants are defined in *<sys/reboot.h>*.

1.7.4. 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

The system abstractions described are:

Directory contexts
: A directory context is a position in the filesystem name space. Operations on files and other named objects in a filesystem are always specified relative to such a context.

Files
: Files are used to store uninterpreted sequences of bytes, which may be *read* and *written* randomly. Pages from files may also be mapped into the process address space. A directory may be read as a file if permitted by the underlying storage facility, though it is usually accessed using *getdirentries* (see section 2.2.3.1). (Local filesystems permit directories to be read, although most NFS implementations do not allow reading of directories.)

Communications domains
: A communications domain represents an interprocess communications environment, such as the communications facilities of the 4.4BSD 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, interrupt generation, output flow control, and editing), magnetic tapes, disks, and other peripherals. They normally 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.

2.1. Generic operations

Many system abstractions support the *read*, *write*, and *ioctl* operations. 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 ssize_t cc; int fd; result void *buf; size_t nbytes;
```

```
cc = write(fd, buf, nbytes);
result ssize_t cc; int fd; void *buf; size_t 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. A return value of 0 is used to indicate an end-of-file condition.

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 only a portion of the provided bytes; the user should resubmit the other bytes in a later request. Error returns because of interrupted or otherwise incomplete operations are possible, in which case no data will have been transferred.

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 {
        char    *iov_base;      /* base of a component */
        size_t  iov_len;        /* length of a component */
};
```

The *iov_base* field should be treated as if its type were ‘‘void *’’ as POSIX and other versions of the structure may use that type. Thus, pointer arithmetic should not use this value without a cast.

The calls using an array of *iovec* structures are:

```
cc = readv(fd, iov, iovlen);
result ssize_t cc; int fd; struct iovec *iov; int iovlen;

cc = writev(fd, iov, iovlen);
result ssize_t 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; u_long 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 used, 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 by <*sys/ioctl.h*>, or more specifically by files it includes.

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 *select* 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 writable.

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 complete immediately. The descriptor may then be *select*'ed for *write* to find out when the operation has been completed. When *select* indicates the descriptor is writable, the operation has completed. Depending on the nature of the descriptor and the operation, additional activity may be started or the new state may be tested.

2.2. Filesystem

2.2.1. Overview

The filesystem abstraction provides access to a hierarchical filesystem structure. The filesystem contains directories (each of which may contain 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. If permitted by the underlying storage mechanism, 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.

2.2.2. Naming

The filesystem 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 NAME_MAX (255) characters excluding null and "/". Each pathname is up to PATH_MAX (1024) characters excluding null.

Each process always has two naming contexts: one for the root directory of the filesystem 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 file name "." in each directory refers to that directory. The file name ".." in each directory refers to the parent directory of that directory. The parent directory of the root of the filesystem is itself.

The calls:

```
chdir(path);
char *path;

fchdir(fd);
int fd;

chroot(path);
char *path;
```

change the current working directory or root directory context of a process. Only the super-user can change the root directory context of a process.

Information about a filesystem that contains a particular file can be obtained using the calls:

```
statfs(path, buf);
char *path; struct statfs *buf;

fstatfs(fd, buf);
int fd; struct statfs *buf;
```

2.2.3. Creation and removal

The filesystem allows directories, files, special devices, and fifos to be created and removed from the filesystem.

2.2.3.1. Directory creation and removal

A directory is created with the *mkdir* system call:

```
mkdir(path, mode);
char *path; mode_t mode;
```

where the mode is defined as for files (see section 2.2.3.2). Directories are removed with the *rmdir* system call:

```
rmdir(path);
char *path;
```

A directory must be empty (other than the entries "." and "..") if it is to be deleted.

Although directories can be read as files, the usual interface is to use the call:

```
getdirentries(fd, buf, nbytes, basep);
int fd; char *buf; int nbytes; long *basep;
```

The *getdirentries* system call returns a canonical array of directory entries in the filesystem independent format described in *<dirent.h>*. Application programs usually use the library routines *opendir*, *readdir*, and *closedir* which provide a more convenient interface than *getdirentries*. The *fts* package is provided for recursive directory traversal.

2.2.3.2. File creation

Files are opened and/or created with the *open* system call:

```
fd = open(path, oflag, mode);
result int fd; char *path; int oflag; mode_t mode;
```

The *path* parameter specifies the name of the file to be opened. The *oflag* parameter must include O_CREAT to cause the file to be created. Bits for *oflag* are defined in *<sys/fcntl.h>*:

```
O_RDONLY    /* open for reading only */
O_WRONLY    /* open for writing only */
O_RDWR      /* open for reading and writing */
O_NONBLOCK  /* no delay */
O_APPEND    /* set append mode */
O_SHLOCK    /* open with shared file lock */
O_EXLOCK    /* open with exclusive file lock */
O_ASYNC     /* signal pgrp when data ready */
O_FSYNC     /* synchronous writes */
O_CREAT     /* create if nonexistent */
O_TRUNC     /* truncate to zero length */
O_EXCL      /* error if already exists */
```

One of O_RDONLY, O_WRONLY and O_RDWR should be specified, indicating what types of operations are desired to be done 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 all writes to be appended to the file. Specifying O_TRUNC causes the file to be truncated when opened. The flag O_CREAT causes the file to be created if it does not exist, owned by the current user and the group of the containing directory. The permissions for the new file are specified in *mode* as the OR of the appropriate permissions as defined in *<sys/stat.h>*:

```
S_IRWXU                 /* RWX for owner */
S_IRUSR                 /* R for owner */
S_IWUSR                 /* W for owner */
S_IXUSR                 /* X for owner */
S_IRWXG                 /* RWX for group */
S_IRGRP                 /* R for group */
S_IWGRP                 /* W for group */
S_IXGRP                 /* X for group */
S_IRWXO                 /* RWX for other */
S_IROTH                 /* R for other */
S_IWOTH                 /* W for other */
S_IXOTH                 /* X for other */
S_ISUID                 /* set user id */
S_ISGID /* set group id */
S_ISTXT /* sticky bit */
```

Historically, the file mode has been used as a four digit octal number. The bottom three digits encode 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. The 7000 bits encode set user ID as 4000, set group ID as 2000, and the sticky bit as 1000. The mode specified to *open* is modified by the process *umask*; permissions specified in the *umask* are cleared in the mode of the created file. The *umask* can be changed with the call:

```
oldmask = umask(newmask);
result mode_t oldmask; mode_t newmask;
```

If the O_EXCL flag is set, and the file already exists, then the *open* will fail without affecting the file in any way. This mechanism provides a simple exclusive access facility. For security reasons, if the O_EXCL flag is set and the file is a symbolic link, the open will fail regardless of the existence of the file referenced by the link. The O_SHLOCK and O_EXLOCK flags allow the file to be atomically *open*'ed and *flock*'ed; see section 2.2.7 for the semantics of *flock* style locks. The O_ASYNC flag enables the SIGIO signal to be sent to the process group of the opening process when I/O is possible, e.g., upon availability of data to be read.

2.2.3.3. Creating references to devices

The filesystem 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 either one of possibly many peripherals of that kind, or special characteristics of the peripheral. Structured devices have all operations done internally in ''block'' quantities while unstructured devices may have input and output done in varying units, and may act as a non-seekable communications channel rather than a random-access device. The *mknod* call creates special entries:

```
mknod(path, mode, dev);
char *path; mode_t mode; dev_t 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.

Fifo's can be created in the filesystem using the call:

```
mkfifo(path, mode);
char *path; mode_t mode;
```

The *mode* parameter is used solely to specify the access permissions of the newly created fifo.

2.2.3.4. Links and renaming

Links allow multiple names for a file to exist. Links exist independently of the file to which they are linked.

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 filesystem. Each link to a file is equivalent, referring to the file independently of any other name. Symbolic links cause string substitution during the pathname interpretation process, and refer to a file name rather than referring directly to a file.

Hard links and symbolic links have different properties. A hard link ensures that the target file will always be accessible, even after its original directory entry is removed; no such guarantee exists for a symbolic link. Unlike hard links, symbolic links can refernce directories and span filesystems boundaries. An *lstat* (see section 2.2.4) call on a hard link will return the information about the file (or directory) that the link references while an *lstat* call on a symbolic link will return information about the link itself. A symbolic link does not have an owner, group, permissions, access and modification times, etc. The only attributes returned from an *lstat* that refer to the symbolic link itself are the file type (S_IFLNK), size, blocks, and link count (always 1). The other attributes are filled in from the directory that contains the link.

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; char *path; result char *buf; int bufsize;
```

This call returns, in *buf*, the string substituted into pathnames passing through *path*. (This string is not NULL terminated.)

Atomic renaming of filesystem resident objects is possible with the *rename* call:

```
rename(oldname, newname);
char *oldname, *newname;
```

where both *oldname* and *newname* must be in the same filesystem. If either *oldname* or *newname* is a directory, then the other also must be a directory for the *rename* to succeed. If *newname* exists and is a directory, then it must be empty.

2.2.3.5. File, device, and fifo removal

A reference to a file, special device or fifo 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. When the last name for a file has been removed, the file may no longer be opened; the file itself is removed once any existing references have been closed.

All current access to a file can be revoked using the call:

```
revoke(path);
char *path;
```

Subsequent operations on any descriptors open at the time of the *revoke* fail, with the exceptions that a *close* call will succeed, and a *read* from a character device file which has been revoked returns a count of zero (end of file). If the file is a special file for a device which is open, the device close function is called as if all open references to the file had been closed. *Open*'s done after the *revoke* may succeed. This call is most useful for revoking access to a terminal line after a hangup in preparation for reuse by a new login session. Access to a controlling terminal is automatically revoked when the session leader for the session exits.

2.2.4. Reading and modifying file attributes

Detailed information about the attributes of a file may be obtained with the calls:

```
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 obtained 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 them and the group ID of the directory in which they were created. The ownership of a file may be changed by either of the calls:

```
chown(path, owner, group);
char *path; uid_t owner; gid_t group;

fchown(fd, owner, group);
int fd, uid_t owner; gid_t group;
```

In addition to ownership, each file has three levels of access protection associated with it. These levels are owner relative, group relative, and other. 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; mode_t mode;

fchmod(fd, mode);
int fd, mode_t mode;
```

where *mode* is a value indicating the new protection of the file, as listed in section 2.2.3.2.

Each file has a set of flags stored as a bit mask associated with it. These flags are returned in the *stat* structure and are set using the calls:

```
chflags(path, flags);
char *path; u_long flags;

fchflags(fd, flags);
int fd; u_long flags;
```

The flags specified are formed by or'ing the following values:

UF_NODUMP	Do not dump the file.
UF_IMMUTABLE	The file may not be changed.
UF_APPEND	The file may only be appended to.
SF_IMMUTABLE	The file may not be changed.
SF_APPEND	The file may only be appended to.

The UF_NODUMP, UF_IMMUTABLE and UF_APPEND flags may be set or unset by either the owner of a file or the super-user. The SF_IMMUTABLE and SF_APPEND flags may only be set or unset by the super-user. They may be set at any time, but normally may only be unset when the system is in single-user mode.

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 file access and modification times.

2.2.5. 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;
```

How is constructed by OR'ing the following bits, defined in *<unistd.h>*:

```
F_OK    /* file exists */
X_OK    /* file is executable/searchable */
W_OK    /* file is writable */
R_OK    /* file is readable */
```

The presence or absence of advisory locks does not affect the result of *access*.

The *pathconf* and *fpathconf* functions provide a method for applications to determine the current value of a configurable system limit or option variable associated with a pathname or file descriptor:

```
ans = pathconf(path, name);
result long ans; char *path; int name;

ans = fpathconf(fd, name);
result long ans; int fd, name;
```

For *pathconf*, the *path* argument is the name of a file or directory. For *fpathconf*, the *fd* argument is an open file descriptor. The *name* argument specifies the system variable to be queried. Symbolic constants for each name value are found in the include file *<unistd.h>*.

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 defined by <*unistd.h*> as one of:

```
SEEK_SET    /* set file offset to offset */
SEEK_CUR    /* set file offset to current plus offset */
SEEK_END    /* set file offset to EOF plus offset */
```

The call ‘‘lseek(fd, 0, SEEK_CUR)’’ returns the current offset into the file.

Files may have ‘‘holes’’ in them. Holes are 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 extended or truncated with either of the calls:

```
truncate(path, length);
char *path; off_t length;

ftruncate(fd, length);
int fd; off_t length;
```

changing the size of the specified file to *length* bytes.

Unless opened with the O_FSYNC flag, writes to files are held for an indeterminate period of time in the system buffer cache. The call:

```
fsync(fd);
int fd;
```

ensures that the contents of a file are committed to disk before returning. This feature is used by applications such as editors that want to ensure the integrity of a new file before continuing.

2.2.7. Locking

The filesystem 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. Byte range locking is available with *fcntl*; see section 1.5.4. The system does not force processes to obey the locks; they are of an advisory nature only.

Locking can be done as part of the *open* call (see section 2.2.3.2) or 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 <*fcntl.h*>:

```
LOCK_SH     /* shared file lock */
LOCK_EX     /* exclusive file lock */
LOCK_NB     /* don't block when locking */
LOCK_UN     /* unlock file */
```

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.

2.2.8. Disk quotas

As an optional facility, each local filesystem can impose limits on a user's or group's disk usage. Two quantities are limited: the total amount of disk space which a user or group may allocate in a filesystem and the total number of files a user or group may create in a filesystem. Quotas are expressed as *hard* limits and *soft* limits. A hard limit is always imposed; if a user or group would exceed a hard limit, the operation which caused the resource request will fail. A soft limit results in the user or group receiving a warning message, but with allocation succeeding. Facilities are provided to turn soft limits into hard limits if a user or group has exceeded a soft limit for an unreasonable period of time.

The *quotactl* call enables, disables and manipulates filesystem quotas:

```
quotactl(path, cmd, id, addr);
char *path; int cmd; int id; char *addr;
```

A quota control command given by cmd operates on the given filename path for the given user ID. The address of an optional command specific data structure, addr, may be given. The supported commands include:

```
Q_QUOTAON     /* enable quotas */
Q_QUOTAOFF    /* disable quotas */
Q_GETQUOTA    /* get limits and usage */
Q_SETQUOTA    /* set limits and usage */
Q_SETUSE      /* set usage */
Q_SYNC        /* sync disk copy of a filesystems quotas */
```

2.2.9. Remote filesystems

There are two system calls intended to help support the remote filesystem implementation. The call:

```
nfssvc(flags, argstructp);
int flags, void *argstructp;
```

is used by the NFS daemons to pass information into and out of the kernel and also to enter the kernel as a server daemon. The flags argument consists of several bits that show what action is to be taken once in the kernel and *argstructp* points to one of three structures depending on which bits are set in flags.

The call:

```
getfh(path, fhp);
char *path; result fhandle_t *fhp;
```

returns a file handle for the specified file or directory in the file handle pointed to by fhp. This file handle can then be used in future calls to NFS to access the file without the need to repeat the pathname translation. This system call is restricted to the superuser.

2.2.10. Other filesystems

The kernel supports many other filesystems. These include:

- The log-structured filesystem. It provides an alternate disk layout than the fast filesystem optimized for writing rather than reading. For further information see the mount_lfs(8) manual page.
- The ISO-standard 9660 filesystem with Rock Ridge extensions used for CD-ROMs. For further information see the mount_cd9660(8) manual page.
- The file descriptor mapping filesystem. For further information see the mount_fdesc(8) manual page.

- The /proc filesystem as an alternative for debuggers. For further information see section 2.5.1 and the mount_procfs(8) manual page.
- The memory-based filesystem, used primarily for fast but ethereal uses such as /tmp. For further information see the mount_mfs(8) manual page.
- The kernel variable filesystem, used as an alternative to *sysctl*. For further information see section 1.7.1 and the mount_kernfs(8) manual page.
- The portal filesystem, used to mount processes in the filesystem. For further information see the mount_portal(8) manual page.
- The uid/gid remapping filesystem, usually layered above NFS filesystems exported to an outside administrative domain. For further information see the mount_umap(8) manual page.
- The union filesystem, used to place a writable filesystem above a read-only filesystem. This filesystem is useful for compiling sources on a CD-ROM without having to copy the CD-ROM contents to writable disk. For further information see the mount_union(8) manual page.

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 (or protocol family) is identified by a manifest constant defined in the file *<sys/socket.h>*. Important standard domains supported by the system are the local (''UNIX'') domain (PF_LOCAL or PF_UNIX) for communication within the system, the ''Internet'' domain (PF_INET) for communication in the DARPA Internet, the ISO family of protocols (PF_ISO and PF_CCITT) for providing a check-off box on the list of your system capabilities, and the ''NS'' domain (PF_NS) for communication using the Xerox Network Systems protocols. 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 of an appropriate type 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

```
SOCK_DGRAM      /* datagram */
SOCK_STREAM     /* virtual circuit */
SOCK_RAW        /* raw socket */
SOCK_RDM        /* reliably-delivered message */
SOCK_SEQPACKET  /* 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. A datagram socket may send messages to and receive messages from multiple peers. 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 or unreliable datagrams. The SOCK_STREAM type models connection-based virtual circuits: two-way byte streams with no record boundaries. Connection setup is required before data communication may begin. The SOCK_SEQPACKET type models a connection-based, full-duplex, reliable, exchange preserving message boundaries; 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 specific *protocol* associated with it. This protocol is used within the domain to provide the semantics required by the socket type. Not all socket types are supported by each domain; support depends on the existence and the implementation of a suitable protocol within the domain. 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.

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;
```

The socket domain and type are as described above, and are specified using the definitions from *<sys/socket.h>*. The protocol may be given as 0, meaning any suitable protocol. One of several possible protocols may be selected using identifiers obtained from a library routine, *getprotobyname*.

An unconnected socket descriptor of a connection-oriented type 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. Datagram sockets need not establish connections before use.

To accept connections or to receive datagrams, a socket must first have a binding to a name (or address) within the communications domain. Such a binding may be established by a *bind* call:

```
bind(s, name, namelen);
int s; struct sockaddr *name; int namelen;
```

Datagram sockets may have default bindings established when first sending data if not explicitly bound earlier. In either case, a socket's bound name may be retrieved with a *getsockname* call:

```
getsockname(s, name, namelen);
int s; result struct sockaddr *name; result int *namelen;
```

while the peer's name can be retrieved with *getpeername*:

```
getpeername(s, name, namelen);
int s; result struct sockaddr *name; result int *namelen;
```

Domains may support sockets with several names.

2.3.1.4. Accepting connections

Once a binding is made to a connection-oriented socket, 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 struct sockaddr *name; result int *anamelen;
```

returns a descriptor for a new, connected, socket from the queue of pending connections on *s*. If no new connections are queued for acceptance, the call will wait for a connection unless non-blocking I/O has been enabled (see section 1.5.4).

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; struct sockaddr *name; int namelen;
```

Although datagram sockets do not establish connections, the *connect* call may be used with such sockets to create an *association* with the foreign address. The address is recorded for use in future *send* calls, which then need not supply destination addresses. Datagrams will be received only from that peer, and asynchronous error reports may be received.

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(domain, type, protocol, sv);
int domain, type, protocol; result int sv[2];
```

Here the returned *sv* descriptors correspond to those obtained with *accept* and *connect*.

The call:

```
pipe(pv);
result int pv[2];
```

creates a pair of SOCK_STREAM sockets in the PF_LOCAL domain, with pv[0] only writable and pv[1] only readable.

2.3.1.6. Sending and receiving data

Messages may be sent from a socket by:

```
cc = sendto(s, msg, len, flags, to, tolen);
result int cc; int s; void *msg; size_t len;
int flags; struct sockaddr *to; int tolen;
```

if the socket is not connected or:

```
cc = send(s, msg, len, flags);
result int cc; int s; void *msg; size_t len; int 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 void *buf; size_t len; int flags;
result struct sockaddr *from; result int *fromlenaddr;
```

and:

```
msglen = recv(s, buf, len, flags);
result int msglen; int s; result void *buf; size_t len; int 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

† 4.4BSD supports *socketpair* creation only in the PF_LOCAL communication domain.

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, sending or receiving high-priority out-of-band messages, or other special requests as follows:

```
MSG_OOB         /* process out-of-band data */
MSG_PEEK        /* peek at incoming message */
MSG_DONTROUTE   /* send without using routing tables */
MSG_EOR         /* data completes record */
MSG_TRUNC       /* data discarded before delivery */
MSG_CTRUNC      /* control data lost before delivery */
MSG_WAITALL     /* wait for full request or error */
MSG_DONTWAIT    /* this message should be nonblocking */
```

2.3.1.7. Scatter/gather and exchanging access rights

It is possible to 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 */
        u_int       msg_namelen;    /* size of address */
        struct      iovec *msg_iov; /* scatter/gather array */
        u_int       msg_iovlen;     /* # elements in msg_iov */
        caddr_t     msg_control;    /* ancillary data */
        u_int       msg_controllen; /* ancillary data buffer len */
        int         msg_flags;      /* flags on received message */
};
```

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 2.1.1. The data in the *msg_control* buffer is composed of an array of variable length messages used for additional information with or about a datagram not expressible by flags. The format is a sequence of message elements headed by *cmsghdr* structures:

```
struct cmsghdr {
        u_int       cmsg_len;       /* data byte count, including hdr */
        int         cmsg_level;     /* originating protocol */
        int         cmsg_type;      /* protocol-specific type */
        u_char      cmsg_data[ ];   /* variable length type specific data */
};
```

The following macros are provided for use with the *msg_control* buffer:

```
CMSG_FIRSTHDR(mhdr)         /* given msghdr, return first cmsghdr */
CMSG_NXTHDR(mhdr, cmsg)     /* given msghdr and cmsghdr, return next cmsghdr */
CMSG_DATA(cmsg)             /* given cmsghdr, return associated data pointer */
```

Access rights to be sent along with the message are specified in one of these *cmsghdr* structures, with level SOL_SOCKET and type SCM_RIGHTS. In the PF_LOCAL domain these are an array of integer

descriptors, copied 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 *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. If the underlying protocol supports unidirectional or bidirectional shutdown, this indication will be passed to the peer. For example, a shutdown for writing might produce an end-of-file condition at the remote end.

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- or protocol-specific facilities. The *getsockopt* and *setsockopt* calls are used to control options:

```
getsockopt(s, level, optname, optval, optlen);
int s, level, optname; result void *optval; result int *optlen;

setsockopt(s, level, optname, optval, optlen);
int s, level, optname; void *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’’ within the protocol family.

2.3.2. PF_LOCAL domain

This section describes briefly the properties of the PF_LOCAL (‘‘UNIX’’) communications domain.

2.3.2.1. Types of sockets

In the local 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 filesystem name space.

2.3.2.3. Access rights transmission

The ability to pass 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 4.4BSD (SMM:18).

2.3.3.1. Socket types and protocols

SOCK_STREAM is supported by the Internet TCP protocol; SOCK_DGRAM by the UDP protocol. Each is layered atop the transport-level Internet Protocol (IP). The Internet Control Message Protocol is implemented atop/beside IP and is accessible via a raw socket. 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 a 32-bit Internet address and a 16-bit port number. Options may be used to provide IP source routing or security options. The 32-bit address is composed of network and host parts; the network part is variable in size and is frequency encoded. The host part may optionally be interpreted as a subnet field plus the host on the subnet; this is is enabled by setting a network address mask at boot time.

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 IP. These interfaces are modeled as SOCK_RAW sockets. Each raw socket is associated with one IP protocol number, and receives all traffic received for that protocol. This approach allows administrative and debugging functions to occur, and enables user-level implementations of special-purpose protocols such as inter-gateway routing protocols.

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 interpretation and editing, and output format and delays.

A terminal may be used as a controlling terminal (login terminal) for a login session. A controlling terminal is associated with a session (see section 1.1.4). A controlling terminal has a foreground process group, which must be a member of the session with which the terminal is associated (see section 1.1.5). Members of the foreground process group are allowed to read from and write to the terminal and change the terminal settings; other process groups from the session may be stopped upon attempts to do these operations.

A session leader allocates a terminal as the controlling terminal for its session using the ioctl

```
ioctl(fd, TIOCSCTTY, NULL);
int fd;
```

Only a session leader may acquire a controlling terminal.

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. These parameters are described in the *termios* structure maintained by the kernel for each terminal line:

```
struct termios {
        tcflag_t    c_iflag;        /* input flags */
        tcflag_t    c_oflag;        /* output flags */
        tcflag_t    c_cflag;        /* control flags */
        tcflag_t    c_lflag;        /* local flags */
        cc_t        c_cc[NCCS];     /* control chars */
        long        c_ispeed;       /* input speed */
        long        c_ospeed;       /* output speed */
};
```

The *termios* structure is set and retrieved using the *tcsetattr* and *tcgetattr* functions.

Two general kinds of input processing are available, determined by whether the terminal device file is in canonical mode or noncanonical mode. Additionally, input characters are processed according to the *c_iflag* and *c_lflag* fields. Such processing can include echoing, which in general means transmitting input characters immediately back to the terminal when they are received from the terminal. Nongraphic ASCII input characters may be echoed as a two-character printable representation, ''^character.''

In canonical mode input processing, terminal input is processed in units of lines. A line is delimited by a newline character (NL), an end-of-file (EOF) character, or an end-of-line (EOL) character. Input is presented on a line-by-line basis. Using this mode means that a read request will not return until an entire line has been typed, or a signal has been received. Also, no matter how many bytes are requested in the read call, at most one line is returned. It is not, however, necessary to read a whole line at once; any number of bytes, even one, may be requested in a read without losing information.

When the terminal is in canonical 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. Certain other characters are also interpreted specially. Flow control is provided by the *stop output* and *start output* control characters. Output may be flushed with the *flush output* character; and the *literal character* may be used to force the following character into the input line, regardless of any special meaning it may have.

In noncanonical mode input processing, input bytes are not assembled into lines, and erase and kill processing does not occur. All input is passed through to the reading process immediately and without interpretation. Signals and flow control may be enabled; here the handler interprets input only by looking for characters that cause interrupts or output flow control; all other characters are made available.

When interrupt characters are being interpreted by the terminal handler they 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.

2.4.1.2. 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, inserting delays after certain standard control characters, and expanding tabs.

2.4.2. Structured devices

Structured 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 random access fashion like ordinary files. Filesystems are normally mounted on 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 debugging

2.5.1. Traditional debugging

Debuggers traditionally use the *ptrace* interface:

```
ptrace(request, pid, addr, data);
int request, pid, *addr, data;
```

This interface provides a means by which a parent process may control the execution of a child process, and examine and change its core image. Its primary use is for the implementation of breakpoint debugging. There are four arguments whose interpretation depends on a request argument. A process being traced behaves normally until it encounters a signal (whether internally generated like ‘‘illegal instruction’’ or externally generated like ‘‘interrupt’’). Then the traced process enters a stopped state and its parent is notified via *wait*. When the child is in the stopped state, its core image can be examined and modified using *ptrace*. Another ptrace request can then cause the child either to terminate or to continue, possibly ignoring the signal.

A more general interface is also provided in 4.4BSD; the *mount_procfs* filesystem attaches an instance of the process name space to the global filesystem name space. The conventional mount point is */proc*. The root of the process filesystem contains an entry for each active process. These processes are visible as directories named by the process' ID. In addition, the special entry *curproc* references the current process. Each directory contains several files, including a *ctl* file. The debugger finds (or creates) the process that it wants to debug and then issues an attach command via the *ctl* file. Further interaction can then be done with the process through the other files provided by the */proc* filesystem.

2.5.2. Kernel tracing

Another facility for debugging programs is provided by the *ktrace* interface:

```
ktrace(tracefile, ops, trpoints, pid);
char *tracefile; int ops, trpoints, pid;
```

Ktrace does kernel trace logging for the specified processes. The kernel operations that are traced include system calls, pathname translations, signal processing, and I/O. This facility can be particularly useful to debug programs for which you do not have the source.

3. Summary of facilities

1 Kernel primitives

1.1 Processes and protection

sethostid	set host identifier
gethostid	get host identifier
sethostname	set host name
gethostname	get host name
getpid	get process identifier
getppid	get parent process identifier
fork	create a new process
vfork	create a new process
exit	terminate a process
wait4	collect exit status of child
execve	execute a new program
getuid	get real user identifier
geteuid	get effective user identifier
getgid	get real group identifier
getegid	get effective group identifier
getgroups	get access group set
setuid	set real, effective, and saved user identifiers
setgid	set real, effective, and saved group identifiers
setgroups	set access group set
seteuid	set effective user identifier
setegid	set effective group identifier
setsid	create a new session
setlogin	set login name
getlogin	get login name
getpgrp	get process group
setpgid	set process group

1.2 Memory management

brk	set data section size
sbrk	change data section size
getpagesize	get system page size
mmap	map files or devices into memory
msync	synchronize a mapped region
munmap	remove a mapping
mprotect	control the protection of pages
madvise	give advise about use of memory
mincore	get advise about use of memory
mlock	lock physical pages in memory
munlock	unlock physical pages in memory
mset	acquire and set a semaphore
mclear	release a semaphore and awaken waiting processes
msleep	wait for a semaphore
mwakeup	awaken process(es) sleeping on a semaphore

1.3 Signals

sigaction	setup software signal handler
sigreturn	return from a signal
kill	send signal to a process
killpg	send signal to a process group
sigprocmask	manipulate current signal mask
sigsuspend	atomically release blocked signals and wait for interrupt

sigpending	get pending signals
sigaltstack	set and/or get signal stack context

1.4 **Timers**

settimeofday	set date and time
gettimeofday	get date and time
adjtime	synchronization of the system clock
setitimer	set value of interval timer
getitimer	get value of interval timer
profil	control process profiling

1.5 **Descriptors**

getdtablesize	get descriptor table size
dup	duplicate an existing file descriptor
dup2	duplicate an existing file descriptor
close	delete a descriptor
select	synchronous I/O multiplexing
fcntl	file control

1.6 **Resource controls**

getpriority	get program scheduling priority
setpriority	set program scheduling priority
getrusage	get information about resource utilization
getrlimit	get maximum system resource consumption
setrlimit	set maximum system resource consumption

1.7 **System operation support**

sysctl	get or set system information
mount	mount a filesystem
getfsstat	get list of all mounted filesystems
swapon	add a swap device for interleaved paging/swapping
unmount	dismount a filesystem
sync	force completion of pending disk writes (flush cache)
reboot	reboot system or halt processor
acct	enable or disable process accounting

2 **System facilities**

2.1 **Generic operations**

read	read input
write	write output
readv	read gathered input
writev	write scattered output
ioctl	control device

2.2 **Filesystem**

chdir	change current working directory
fchdir	change current working directory
chroot	change root directory
statfs	get file system statistics
fstatfs	get file system statistics
mkdir	make a directory file
rmdir	remove a directory file
getdirentries	get directory entries in a filesystem independent format
open	open or create a file for reading or writing
umask	set file creation mode mask
mknod	make a special file node
mkfifo	make a fifo file
link	make a hard file link

symlink	make a symbolic link to a file
readlink	read value of a symbolic link
rename	change the name of a file
unlink	remove directory entry
revoke	revoke file access
stat	get file status
fstat	get file status
lstat	get file status
chown	change owner and group of a file
fchown	change owner and group of a file
chmod	change mode of file
fchmod	change mode of file
chflags	set file flags
fchflags	set file flags
utimes	set file access and modification times
access	check access permissions of a file or pathname
pathconf	get configurable pathname variables
fpathconf	get configurable pathname variables
lseek	reposition read/write file offset
truncate	truncate a file to a specified length
ftruncate	truncate a file to a specified length
fsync	synchronize in-core state of a file with that on disk
flock	apply or remove an advisory lock on an open file
quotactl	manipulate filesystem quotas
nfssvc	NFS services
getfh	get file handle

2.3 **Interprocess communications**

socket	create an endpoint for communication
bind	bind a name to a socket
getsockname	get socket name
getpeername	get name of connected peer
listen	listen for connections on a socket
accept	accept a connection on a socket
connect	initiate a connection on a socket
socketpair	create a pair of connected sockets
pipe	create descriptor pair for interprocess communication
sendto	send a message from a socket
send	send a message from a socket
recvfrom	receive a message from a socket
recv	receive a message from a socket
sendmsg	send a message from a socket
recvmsg	receive a message from a socket
shutdown	shut down part of a full-duplex connection
getsockopt	get options on socket
setsockopt	set options on socket

2.4 **Terminals and Devices**

2.5 **Process debugging**

ptrace	process trace
ktrace	process tracing

3 **Summary of facilities**

Contents

Notation and Types 4

1 **Kernel primitives** 4

1.1 **Processes and protection** 5
1.1.1 Host identifiers 5
1.1.2 Process identifiers 5
1.1.3 Process creation and termination 5
1.1.4 User and group IDs 6
1.1.5 Sessions 7
1.1.6 Process groups 7

1.2 **Memory management** 8
1.2.1 Text, data, and stack 8
1.2.2 Mapping pages 8
1.2.3 Page protection control 10
1.2.4 Giving and getting advice 10
1.2.5 Synchronization primitives 11

1.3 **Signals** 11
1.3.1 Overview 11
1.3.2 Signal types 12
1.3.3 Signal handlers 12
1.3.4 Sending signals 13
1.3.5 Protecting critical sections 14
1.3.6 Signal stacks 14

1.4 **Timers** 14
1.4.1 Real time 14
1.4.2 Interval time 15

1.5 **Descriptors** 16
1.5.1 The reference table 16
1.5.2 Descriptor properties 16
1.5.3 Managing descriptor references 17
1.5.4 Multiplexing requests 17

1.6 **Resource controls** 18
1.6.1 Process priorities 18
1.6.2 Resource utilization 19
1.6.3 Resource limits 19

1.7 **System operation support** 20
1.7.1 Monitoring system operation 20
1.7.2 Bootstrap operations 21
1.7.3 Shutdown operations 21
1.7.4 Accounting 21

2 **System facilities** 22

2.1 **Generic operations** 22
2.1.1 Read and write 22
2.1.2 Input/output control 23
2.1.3 Non-blocking and asynchronous operations 23

2.2 **Filesystem** 24
2.2.1 Overview 24
2.2.2 Naming 24
2.2.3 Creation and removal 24

2.2.3.1Directory creation and removal25
2.2.3.2File creation25
2.2.3.3Creating references to devices26
2.2.3.4Links and renaming27
2.2.3.5File, device, and fifo removal27
2.2.4Reading and modifying file attributes28
2.2.5Checking accessibility29
2.2.6Extension and truncation29
2.2.7Locking30
2.2.8Disk quotas30
2.2.9Remote filesystems31
2.2.10Other filesystems31
2.3**Interprocess communications**32
2.3.1Interprocess communication primitives32
2.3.1.1Communication domains32
2.3.1.2Socket types and protocols32
2.3.1.3Socket creation, naming and service establishment33
2.3.1.4Accepting connections33
2.3.1.5Making connections34
2.3.1.6Sending and receiving data34
2.3.1.7Scatter/gather and exchanging access rights35
2.3.1.8Using read and write with sockets36
2.3.1.9Shutting down halves of full-duplex connections36
2.3.1.10Socket and protocol options36
2.3.2PF_LOCAL domain36
2.3.2.1Types of sockets36
2.3.2.2Naming36
2.3.2.3Access rights transmission37
2.3.3INTERNET domain37
2.3.3.1Socket types and protocols37
2.3.3.2Socket naming37
2.3.3.3Access rights transmission37
2.3.3.4Raw access37
2.4**Terminals and Devices**37
2.4.1Terminals37
2.4.1.1Terminal input38
2.4.1.2Terminal output38
2.4.2Structured devices38
2.4.3Unstructured devices39
2.5**Process debugging**39
2.5.1Traditional debugging39
2.5.2Kernel tracing39
3**Summary of facilities**40

The C Programming Language - Reference Manual

Dennis M. Ritchie

AT&T Bell Laboratories
Murray Hill, NJ 07974

This manual is a reprint, with updates to the current C standard, from *The C Programming Language*, by Brian W. Kernighan and Dennis M. Richie, Prentice-Hall, Inc., 1978.

1. Introduction

This manual describes the C language on the DEC PDP-11†, the DEC VAX-11, and the AT&T 3B 20‡. Where differences exist, it concentrates on the VAX, but tries to point out implementation-dependent details. With few exceptions, these dependencies follow directly from the underlying properties of the hardware; the various compilers are generally quite compatible.

2. Lexical Conventions

There are six classes of tokens - identifiers, keywords, constants, strings, operators, and other separators. Blanks, tabs, new-lines, and comments (collectively, ‘‘white space’’) as described below are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

2.1. Comments

The characters **/*** introduce a comment which terminates with the characters ∗**/**. Comments do not nest.

2.2. Identifiers (Names)

An identifier is a sequence of letters and digits. The first character must be a letter. The underscore (_) counts as a letter. Uppercase and lowercase letters are different. Although there is no limit on the length of a name, only initial characters are significant: at least eight characters of a non-external name, and perhaps fewer for external names. Moreover, some implementations may collapse case distinctions for external names. The external name sizes include:

PDP-11	7 characters, 2 cases
VAX-11	>100 characters, 2 cases
AT&T 3B 20	>100 characters, 2 cases

---

† DEC PDP-11, and DEC VAX-11 are trademarks of Digital Equipment Corporation.

‡ 3B 20 is a trademark of AT&T.

2.3. Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise:

auto	**do**	**for**	**return**	**typedef**
break	**double**	**goto**	**short**	**union**
case	**else**	**if**	**sizeof**	**unsigned**
char	**enum**	**int**	**static**	**void**
continue	**external**	**long**	**struct**	**while**
default	**float**	**register**	**switch**	

Some implementations also reserve the words **fortran, asm, gfloat, hfloat** and **quad**

2.4. Constants

There are several kinds of constants. Each has a type; an introduction to types is given in ''NAMES.'' Hardware characteristics that affect sizes are summarized in ''Hardware Characteristics'' under ''LEXICAL CONVENTIONS.''

2.4.1. Integer Constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with **0** (digit zero). An octal constant consists of the digits **0** through **7** only. A sequence of digits preceded by **0x** or **0X** (digit zero) is taken to be a hexadecimal integer. The hexadecimal digits include **a** or **A** through **f** or **F** with values 10 through 15. Otherwise, the integer constant is taken to be decimal. A decimal constant whose value exceeds the largest signed machine integer is taken to be **long**; an octal or hex constant which exceeds the largest unsigned machine integer is likewise taken to be **long**. Otherwise, integer constants are **int**.

2.4.2. Explicit Long Constants

A decimal, octal, or hexadecimal integer constant immediately followed by **l** (letter ell) or **L** is a long constant. As discussed below, on some machines integer and long values may be considered identical.

2.4.3. Character Constants

A character constant is a character enclosed in single quotes, as in '**x**'. The value of a character constant is the numerical value of the character in the machine's character set.

Certain nongraphic characters, the single quote (**'**) and the backslash (\), may be represented according to the following table of escape sequences:

new-line	NL (LF)	\n	
horizontal tab	HT	\t	
vertical tab	VT	\v	
backspace	BS	\b	
carriage return	CR	\r	
form feed	FF	\f	
backslash	\	\\	
single quote	'	\'	
bit pattern	*ddd*	*ddd*	

The escape *ddd* consists of the backslash followed by 1, 2, or 3 octal digits which are taken to specify the value of the desired character. A special case of this construction is **\0** (not followed by a digit), which indicates the character **NUL**. If the character following a backslash is not one of those specified, the behavior is undefined. A new-line character is illegal in a character constant. The type of a character constant is **int**.

2.4.4. Floating Constants

A floating constant consists of an integer part, a decimal point, a fraction part, an **e** or **E**, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing. Either the decimal point or the **e** and the exponent (not both) may be missing. Every floating constant has type **double**.

2.4.5. Enumeration Constants

Names declared as enumerators (see ''Structure, Union, and Enumeration Declarations'' under ''DECLARATIONS'') have type **int**.

2.5. Strings

A string is a sequence of characters surrounded by double quotes, as in **"..."**. A string has type ''array of **char**'' and storage class **static** (see ''NAMES'') and is initialized with the given characters. The compiler places a null byte (**\0**) at the end of each string so that programs which scan the string can find its end. In a string, the double quote character (**"**) must be preceded by a \; in addition, the same escapes as described for character constants may be used.

A \ and the immediately following new-line are ignored. All strings, even when written identically, are distinct.

2.6. Hardware Characteristics

The following figure summarize certain hardware properties that vary from machine to machine.

	DEC PDP–11 (ASCII)	**DEC VAX-11** (ASCII)	**AT&T 3B** (ASCII)
char	8 bits	8 bits	8bits
int	16	32	32
short	16	16	16
long	32	32	32
float	32	32	32
double	64	64	64
float range	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$
double range	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$	$\pm 10^{\pm 308}$

3. Syntax Notation

Syntactic categories are indicated by *italic* type and literal words and characters in **bold** type. Alternative categories are listed on separate lines. An optional terminal or nonterminal symbol is indicated by the subscript ''opt,'' so that

$$\{\ expression_{opt}\ \}$$

indicates an optional expression enclosed in braces. The syntax is summarized in ''SYNTAX SUMMARY''.

4. Names

The C language bases the interpretation of an identifier upon two attributes of the identifier – its *storage class* and its *type*. The storage class determines the location and lifetime of the storage associated with an identifier; the type determines the meaning of the values found in the identifier's storage.

4.1. Storage Class

There are four declarable storage classes: Automatic Static External Register.

Automatic variables are local to each invocation of a block (see ''Compound Statement or Block'' in ''STATEMENTS'') and are discarded upon exit from the block. Static variables are local to a block but retain their values upon reentry to a block even after control has left the block. External variables exist and retain their values throughout the execution of the entire program and may be used for communication between functions, even separately compiled functions. Register variables are (if possible) stored in the fast registers of the machine; like automatic variables, they are local to each block and disappear on exit from the block.

4.2. Type

The C language supports several fundamental types of objects. Objects declared as characters (**char**) are large enough to store any member of the implementation's character set. If a genuine character from that character set is stored in a **char** variable, its value is equivalent to the integer code for that character. Other quantities may be stored into character variables, but the implementation is machine dependent. In particular, **char** may be signed or unsigned by default.

Up to three sizes of integer, declared **short int**, **int**, and **long int**, are available. Longer integers provide no less storage than shorter ones, but the implementation may make either short integers or long integers, or both, equivalent to plain integers. ''Plain'' integers have the natural size suggested by the host machine architecture. The other sizes are provided to meet special needs.

The properties of **enum** types (see ''Structure, Union, and Enumeration Declarations'' under ''DECLARATIONS'') are identical to those of some integer types. The implementation may use the range of values to determine how to allocate storage.

Unsigned integers, declared **unsigned,** obey the laws of arithmetic modulo 2^n where n is the number of bits in the representation. (On the PDP-11, unsigned long quantities are not supported.)

Single-precision floating point (**float**) and double precision floating point (**double**) may be synonymous in some implementations.

Because objects of the foregoing types can usefully be interpreted as numbers, they will be referred to as *arithmetic* types. **Char**, **int** of all sizes whether **unsigned** or not, and **enum** will collectively be called *integral* types. The **float** and **double** types will collectively be called *floating* types.

The **void** type specifies an empty set of values. It is used as the type returned by functions that generate no value.

Besides the fundamental arithmetic types, there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways: *Arrays* of objects of most types *Functions* which return objects of a given type *Pointers* to objects of a given type *Structures* containing a sequence of objects of various types *Unions* capable of containing any one of several objects of various types.

In general these methods of constructing objects can be applied recursively.

5. Objects and Lvalues

An *object* is a manipulatable region of storage. An *lvalue* is an expression referring to an object. An obvious example of an lvalue expression is an identifier. There are operators which yield lvalues: for example, if **E** is an expression of pointer type, then ∗**E** is an lvalue expression referring to the object to which **E** points. The name ''lvalue'' comes from the assignment expression **E1 = E2** in which the left operand **E1** must be an lvalue expression. The discussion of each operator below indicates whether it expects lvalue operands and whether it yields an lvalue.

6. Conversions

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This part explains the result to be expected from such conversions. The conversions demanded by most ordinary operators are summarized under ''Arithmetic Conversions.'' The summary will be supplemented as required by the discussion of each operator.

6.1. Characters and Integers

A character or a short integer may be used wherever an integer may be used. In all cases the value is converted to an integer. Conversion of a shorter integer to a longer preserves sign. Whether or not sign-extension occurs for characters is machine dependent, but it is guaranteed that a member of the standard character set is non-negative. Of the machines treated here, only the PDP-11 and VAX-11 sign-extend. On these machines, **char** variables range in value from −128 to 127. The more explicit type **unsigned char** forces the values to range from 0 to 255.

On machines that treat characters as signed, the characters of the ASCII set are all non-negative. However, a character constant specified with an octal escape suffers sign extension and may appear negative; for example, **'\377'** has the value **−1**.

When a longer integer is converted to a shorter integer or to a **char,** it is truncated on the left. Excess bits are simply discarded.

6.2. Float and Double

All floating arithmetic in C is carried out in double precision. Whenever a **float** appears in an expression it is lengthened to **double** by zero padding its fraction. When a **double** must be converted to **float**, for example by an assignment, the **double** is rounded before truncation to **float** length. This result is undefined if it cannot be represented as a float. On the VAX, the compiler can be directed to use single percision for expressions containing only float and interger operands.

6.3. Floating and Integral

Conversions of floating values to integral type are rather machine dependent. In particular, the direction of truncation of negative numbers varies. The result is undefined if it will not fit in the space provided.

Conversions of integral values to floating type are well behaved. Some loss of accuracy occurs if the destination lacks sufficient bits.

6.4. Pointers and Integers

An expression of integral type may be added to or subtracted from a pointer; in such a case, the first is converted as specified in the discussion of the addition operator. Two pointers to objects of the same type may be subtracted; in this case, the result is converted to an integer as specified in the discussion of the subtraction operator.

6.5. Unsigned

Whenever an unsigned integer and a plain integer are combined, the plain integer is converted to unsigned and the result is unsigned. The value is the least unsigned integer congruent to the signed integer (modulo $2^{word\text{-}size}$). In a 2's complement representation, this conversion is conceptual; and there is no actual change in the bit pattern.

When an unsigned **short** integer is converted to **long**, the value of the result is the same numerically as that of the unsigned integer. Thus the conversion amounts to padding with zeros on the left.

6.6. Arithmetic Conversions

A great many operators cause conversions and yield result types in a similar way. This pattern will be called the ''usual arithmetic conversions.'' First, any operands of type **char** or **short** are converted to **int**, and any operands of type **unsigned char** or **unsigned short** are converted to **unsigned int**. Then, if either operand is **double,** the other is converted to **double** and that is the type of the result. Otherwise, if either operand is **unsigned long**, the other is converted to **unsigned long** and that is the type of the result. Otherwise, if either operand is **long**, the other is converted to **long** and that is the type of the result. Otherwise, if one operand is **long**, and the other is **unsigned int**, they are both converted to **unsigned long** and that is the type of the result. Otherwise, if either operand is **unsigned,** the other is converted to **unsigned** and that is the type of the result. Otherwise, both operands must be **int**, and that is the type of the result.

6.7. Void

The (nonexistent) value of a **void** object may not be used in any way, and neither explicit nor implicit conversion may be applied. Because a void expression denotes a nonexistent value, such an expression may be used only as an expression statement (see ''Expression Statement'' under ''STATEMENTS'') or as the left operand of a comma expression (see ''Comma Operator'' under ''EXPRESSIONS'').

An expression may be converted to type **void** by use of a cast. For example, this makes explicit the discarding of the value of a function call used as an expression statement.

7. Expressions

The precedence of expression operators is the same as the order of the major subsections of this section, highest precedence first. Thus, for example, the expressions referred to as the operands of + (see ''Additive Operators'') are those expressions defined under ''Primary Expressions'', ''Unary Operators'', and ''Multiplicative Operators''. Within each subpart, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The precedence and associativity of all the expression operators are summarized in the grammar of ''SYNTAX SUMMARY''.

Otherwise, the order of evaluation of expressions is undefined. In particular, the compiler considers itself free to compute subexpressions in the order it believes most efficient even if the subexpressions involve side effects. The order in which subexpression evaluation takes place is unspecified. Expressions involving a commutative and associative operator (*, +, **&**, |, ^) may be rearranged arbitrarily even in the presence of parentheses; to force a particular order of evaluation, an explicit temporary must be used.

The handling of overflow and divide check in expression evaluation is undefined. Most existing implementations of C ignore integer overflows; treatment of division by 0 and all floating-point exceptions varies between machines and is usually adjustable by a library function.

7.1. Primary Expressions

Primary expressions involving **.**, **−>**, subscripting, and function calls group left to right.

primary-expression:
identifier
constant
string
(expression)
primary-expression [expression]
primary-expression (expression-list$_{opt}$ *)*
primary-expression . identifier
primary-expression −> identifier

expression-list:
expression
expression-list , expression

An identifier is a primary expression provided it has been suitably declared as discussed below. Its type is specified by its declaration. If the type of the identifier is ''array of ...'', then the value of the identifier expression is a pointer to the first object in the array; and the type of the expression is ''pointer to ...''. Moreover, an array identifier is not an lvalue expression. Likewise, an identifier which is declared ''function returning ...'', when used except in the function-name position of a call, is converted to ''pointer to function returning ...''.

A constant is a primary expression. Its type may be **int**, **long**, or **double** depending on its form. Character constants have type **int** and floating constants have type **double**.

A string is a primary expression. Its type is originally ''array of **char**'', but following the same rule given above for identifiers, this is modified to ''pointer to **char**'' and the result is a pointer to the first character in the string. (There is an exception in certain initializers; see ''Initialization'' under ''DECLARATIONS.'')

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

A primary expression followed by an expression in square brackets is a primary expression. The intuitive meaning is that of a subscript. Usually, the primary expression has type "pointer to ...", the subscript expression is **int**, and the type of the result is "...". The expression **E1[E2]** is identical (by definition) to ∗**((E1)+E2))**. All the clues needed to understand this notation are contained in this subpart together with the discussions in "Unary Operators" and "Additive Operators" on identifiers, ∗ and + respectively. The implications are summarized under "Arrays, Pointers, and Subscripting" under "TYPES REVISITED."

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. The primary expression must be of type "function returning ...," and the result of the function call is of type "...". As indicated below, a hitherto unseen identifier followed immediately by a left parenthesis is contextually declared to represent a function returning an integer; thus in the most common case, integer-valued functions need not be declared.

Any actual arguments of type **float** are converted to **double** before the call. Any of type **char** or **short** are converted to **int**. Array names are converted to pointers. No other conversions are performed automatically; in particular, the compiler does not compare the types of actual arguments with those of formal arguments. If conversion is needed, use a cast; see "Unary Operators" and "Type Names" under "DECLARATIONS."

In preparing for the call to a function, a copy is made of each actual parameter. Thus, all argument passing in C is strictly by value. A function may change the values of its formal parameters, but these changes cannot affect the values of the actual parameters. It is possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points. An array name is a pointer expression. The order of evaluation of arguments is undefined by the language; take note that the various compilers differ. Recursive calls to any function are permitted.

A primary expression followed by a dot followed by an identifier is an expression. The first expression must be a structure or a union, and the identifier must name a member of the structure or union. The value is the named member of the structure or union, and it is an lvalue if the first expression is an lvalue.

A primary expression followed by an arrow (built from − and >) followed by an identifier is an expression. The first expression must be a pointer to a structure or a union and the identifier must name a member of that structure or union. The result is an lvalue referring to the named member of the structure or union to which the pointer expression points. Thus the expression **E1−>MOS** is the same as **(∗E1).MOS**. Structures and unions are discussed in "Structure, Union, and Enumeration Declarations" under "DECLARATIONS."

7.2. Unary Operators

Expressions with unary operators group right to left.

unary-expression:
- ∗ *expression*
- & *lvalue*
- − *expression*
- *! expression*
- ~ *expression*
- ++ *lvalue*
- −−*lvalue*
- *lvalue* ++
- *lvalue* −−
- *(type-name) expression*
- sizeof *expression*
- sizeof *(type-name)*

The unary ∗ operator means *indirection* ; the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points. If the type of the expression is "pointer to ...," the type of the result is "...".

The result of the unary **&** operator is a pointer to the object referred to by the lvalue. If the type of the lvalue is "...", the type of the result is "pointer to ...".

The result of the unary − operator is the negative of its operand. The usual arithmetic conversions are performed. The negative of an unsigned quantity is computed by subtracting its value from 2^n where n is the number of bits in the corresponding signed type.

There is no unary + operator.

The result of the logical negation operator **!** is one if the value of its operand is zero, zero if the value of its operand is nonzero. The type of the result is **int**. It is applicable to any arithmetic type or to pointers.

The **~** operator yields the one's complement of its operand. The usual arithmetic conversions are performed. The type of the operand must be integral.

The object referred to by the lvalue operand of prefix ++ is incremented. The value is the new value of the operand but is not an lvalue. The expression **++x** is equivalent to **x=x+1**. See the discussions "Additive Operators" and "Assignment Operators" for information on conversions.

The lvalue operand of prefix −− is decremented analogously to the prefix ++ operator.

When postfix ++ is applied to an lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is incremented in the same manner as for the prefix ++ operator. The type of the result is the same as the type of the lvalue expression.

When postfix −− is applied to an lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is decremented in the manner as for the prefix −− operator. The type of the result is the same as the type of the lvalue expression.

An expression preceded by the parenthesized name of a data type causes conversion of the value of the expression to the named type. This construction is called a *cast*. Type names are described in "Type Names" under "Declarations."

The **sizeof** operator yields the size in bytes of its operand. (A *byte* is undefined by the language except in terms of the value of **sizeof**. However, in all existing implementations, a byte is the space required to hold a **char.**) When applied to an array, the result is the total number of bytes in the array. The size is determined from the declarations of the objects in the expression. This expression is semantically an **unsigned** constant and may be used anywhere a constant is required. Its major use is in communication with routines like storage allocators and I/O systems.

The **sizeof** operator may also be applied to a parenthesized type name. In that case it yields the size in bytes of an object of the indicated type.

The construction **sizeof(***type* **)** is taken to be a unit, so the expression **sizeof(***type* **)-2** is the same as **(sizeof(***type* **))-2**.

7.3. Multiplicative Operators

The multiplicative operators ∗, **/**, and **%** group left to right. The usual arithmetic conversions are performed.

multiplicative expression:
expression ∗ *expression*
expression **/** *expression*
expression **%** *expression*

The binary ∗ operator indicates multiplication. The ∗ operator is associative, and expressions with several multiplications at the same level may be rearranged by the compiler. The binary **/** operator indicates division.

The binary **%** operator yields the remainder from the division of the first expression by the second. The operands must be integral.

When positive integers are divided, truncation is toward 0; but the form of truncation is machine-dependent if either operand is negative. On all machines covered by this manual, the remainder has the same sign as the dividend. It is always true that **(a/b)∗b + a%b** is equal to **a** (if **b** is not 0).

7.4. Additive Operators

The additive operators + and − group left to right. The usual arithmetic conversions are performed. There are some additional type possibilities for each operator.

additive-expression:
expression + *expression*
expression − *expression*

The result of the + operator is the sum of the operands. A pointer to an object in an array and a value of any integral type may be added. The latter is in all cases converted to an address offset by multiplying it by the length of the object to which the pointer points. The result is a pointer of the same type as the original pointer which points to another object in the same array, appropriately offset from the original object. Thus if **P** is a pointer to an object in an array, the expression **P**+**1** is a pointer to the next object in the array. No further type combinations are allowed for pointers.

The + operator is associative, and expressions with several additions at the same level may be rearranged by the compiler.

The result of the − operator is the difference of the operands. The usual arithmetic conversions are performed. Additionally, a value of any integral type may be subtracted from a pointer, and then the same conversions for addition apply.

If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an **int** representing the number of objects separating the pointed-to objects. This conversion will in general give unexpected results unless the pointers point to objects in the same array, since pointers, even to objects of the same type, do not necessarily differ by a multiple of the object length.

7.5. Shift Operators

The shift operators **<<** and **>>** group left to right. Both perform the usual arithmetic conversions on their operands, each of which must be integral. Then the right operand is converted to **int**; the type of the result is that of the left operand. The result is undefined if the right operand is negative or greater than or equal to the length of the object in bits. On the VAX a negative right operand is interpreted as reversing the direction of the shift.

shift-expression:
expression << *expression*
expression >> *expression*

The value of **E1<<E2** is **E1** (interpreted as a bit pattern) left-shifted **E2** bits. Vacated bits are 0 filled. The value of **E1>>E2** is **E1** right-shifted **E2** bit positions. The right shift is guaranteed to be logical (0 fill) if **E1** is **unsigned**; otherwise, it may be arithmetic.

7.6. Relational Operators

The relational operators group left to right.

relational-expression:
expression < *expression*
expression > *expression*
expression <= *expression*
expression >= *expression*

The operators **<** (less than), **>** (greater than), **<=** (less than or equal to), and **>=** (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is **int**. The usual arithmetic conversions are performed. Two pointers may be compared; the result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is portable only when the pointers point to objects in the same array.

7.7. Equality Operators

equality-expression:
expression == expression
expression != expression

The **==** (equal to) and the **!=** (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus **a<b == c<d** is 1 whenever **a<b** and **c<d** have the same truth value).

A pointer may be compared to an integer only if the integer is the constant 0. A pointer to which 0 has been assigned is guaranteed not to point to any object and will appear to be equal to 0. In conventional usage, such a pointer is considered to be null.

7.8. Bitwise AND Operator

and-expression:
expression & expression

The **&** operator is associative, and expressions involving **&** may be rearranged. The usual arithmetic conversions are performed. The result is the bitwise AND function of the operands. The operator applies only to integral operands.

7.9. Bitwise Exclusive OR Operator

exclusive-or-expression:
expression ^ expression

The **^** operator is associative, and expressions involving **^** may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of the operands. The operator applies only to integral operands.

7.10. Bitwise Inclusive OR Operator

inclusive-or-expression:
expression | expression

The **|** operator is associative, and expressions involving **|** may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of its operands. The operator applies only to integral operands.

7.11. Logical AND Operator

logical-and-expression:
expression && expression

The **&&** operator groups left to right. It returns 1 if both its operands evaluate to nonzero, 0 otherwise. Unlike **&**, **&&** guarantees left to right evaluation; moreover, the second operand is not evaluated if the first operand is 0.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always **int**.

7.12. Logical OR Operator

logical-or-expression:
expression || expression

The **||** operator groups left to right. It returns 1 if either of its operands evaluates to nonzero, 0 otherwise. Unlike **|**, **||** guarantees left to right evaluation; moreover, the second operand is not evaluated if the value of the

first operand is nonzero.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always **int**.

7.13. Conditional Operator

conditional-expression:
expression ? expression : expression

Conditional expressions group right to left. The first expression is evaluated; and if it is nonzero, the result is the value of the second expression, otherwise that of third expression. If possible, the usual arithmetic conversions are performed to bring the second and third expressions to a common type. If both are structures or unions of the same type, the result has the type of the structure or union. If both pointers are of the same type, the result has the common type. Otherwise, one must be a pointer and the other the constant 0, and the result has the type of the pointer. Only one of the second and third expressions is evaluated.

7.14. Assignment Operators

There are a number of assignment operators, all of which group right to left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place. The two parts of a compound assignment operator are separate tokens.

assignment-expression:
lvalue = expression
lvalue += expression
lvalue −= expression
*lvalue *= expression*
lvalue /= expression
lvalue %= expression
lvalue >>= expression
lvalue <<= expression
lvalue &= expression
lvalue ^= expression
lvalue |= expression

In the simple assignment with **=**, the value of the expression replaces that of the object referred to by the lvalue. If both operands have arithmetic type, the right operand is converted to the type of the left preparatory to the assignment. Second, both operands may be structures or unions of the same type. Finally, if the left operand is a pointer, the right operand must in general be a pointer of the same type. However, the constant 0 may be assigned to a pointer; it is guaranteed that this value will produce a null pointer distinguishable from a pointer to any object.

The behavior of an expression of the form **E1** *op* **= E2** may be inferred by taking it as equivalent to **E1 = E1** *op* **(E2)**; however, **E1** is evaluated only once. In **+=** and **−=**, the left operand may be a pointer; in which case, the (integral) right operand is converted as explained in "Additive Operators." All right operands and all nonpointer left operands must have arithmetic type.

7.15. Comma Operator

comma-expression:
expression , expression

A pair of expressions separated by a comma is evaluated left to right, and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left to right. In contexts where comma is given a special meaning, e.g., in lists of actual arguments to functions (see "Primary Expressions") and lists of initializers (see "Initialization" under "DECLARATIONS"), the

comma operator as described in this subpart can only appear in parentheses. For example,

f(a, (t=3, t+2), c)

has three arguments, the second of which has the value 5.

8. Declarations

Declarations are used to specify the interpretation which C gives to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations have the form

declaration:
decl-specifiers *declarator-list*$_{opt}$;

The declarators in the declarator-list contain the identifiers being declared. The decl-specifiers consist of a sequence of type and storage class specifiers.

decl-specifiers:
type-specifier *decl-specifiers*$_{opt}$
sc-specifier *decl-specifiers*$_{opt}$

The list must be self-consistent in a way described below.

8.1. Storage Class Specifiers

The sc-specifiers are:

sc-specifier:
auto
static
extern
register
typedef

The **typedef** specifier does not reserve storage and is called a ‘‘storage class specifier’’ only for syntactic convenience. See ‘‘Typedef’’ for more information. The meanings of the various storage classes were discussed in ‘‘Names.’’

The **auto**, **static**, and **register** declarations also serve as definitions in that they cause an appropriate amount of storage to be reserved. In the **extern** case, there must be an external definition (see ‘‘External Definitions’’) for the given identifiers somewhere outside the function in which they are declared.

A **register** declaration is best thought of as an **auto** declaration, together with a hint to the compiler that the variables declared will be heavily used. Only the first few such declarations in each function are effective. Moreover, only variables of certain types will be stored in registers; on the PDP-11, they are **int** or pointer. One other restriction applies to register variables: the address-of operator **&** cannot be applied to them. Smaller, faster programs can be expected if register declarations are used appropriately, but future improvements in code generation may render them unnecessary.

At most, one sc-specifier may be given in a declaration. If the sc-specifier is missing from a declaration, it is taken to be **auto** inside a function, **extern** outside. Exception: functions are never automatic.

8.2. Type Specifiers

The type-specifiers are

type-specifier:
struct-or-union-specifier
typedef-name
enum-specifier
basic-type-specifier:
basic-type
basic-type basic-type-specifiers
basic-type:
char
short
int
long
unsigned
float
double
void

At most one of the words **long** or **short** may be specified in conjunction with **int**; the meaning is the same as if **int** were not mentioned. The word **long** may be specified in conjunction with **float**; the meaning is the same as **double**. The word **unsigned** may be specified alone, or in conjunction with **int** or any of its short or long varieties, or with **char**.

Otherwise, at most on type-specifier may be given in a declaration. In particular, adjectival use of **long**, **short**, or **unsigned** is not permitted with **typedef** names. If the type-specifier is missing from a declaration, it is taken to be **int**.

Specifiers for structures, unions, and enumerations are discussed in ''Structure, Union, and Enumeration Declarations.'' Declarations with **typedef** names are discussed in ''Typedef.''

8.3. Declarators

The declarator-list appearing in a declaration is a comma-separated sequence of declarators, each of which may have an initializer.

declarator-list:
init-declarator
init-declarator , declarator-list

init-declarator:
declarator initializer$_{opt}$

Initializers are discussed in ''Initialization''. The specifiers in the declaration indicate the type and storage class of the objects to which the declarators refer. Declarators have the syntax:

declarator:
identifier
(declarator)
** declarator*
declarator ()
declarator [constant-expression$_{opt}$ *]*

The grouping is the same as in expressions.

8.4. Meaning of Declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type and storage class.

Each declarator contains exactly one identifier; it is this identifier that is declared. If an unadorned identifier appears as a declarator, then it has the type indicated by the specifier heading the declaration.

A declarator in parentheses is identical to the unadorned declarator, but the binding of complex declarators may be altered by parentheses. See the examples below.

Now imagine a declaration

T D1

where **T** is a type-specifier (like **int**, etc.) and **D1** is a declarator. Suppose this declaration makes the identifier have type "... **T** ," where the "..." is empty if **D1** is just a plain identifier (so that the type of **x** in **'int x"** is just **int**). Then if **D1** has the form

∗**D**

the type of the contained identifier is "... pointer to **T** ."

If **D1** has the form

D ( )

then the contained identifier has the type "... function returning **T**."

If **D1** has the form

D [*constant-expression* **]**

or

D [ ]

then the contained identifier has type "... array of **T**." In the first case, the constant expression is an expression whose value is determinable at compile time , whose type is **int**, and whose value is positive. (Constant expressions are defined precisely in "Constant Expressions.") When several "array of" specifications are adjacent, a multidimensional array is created; the constant expressions which specify the bounds of the arrays may be missing only for the first member of the sequence. This elision is useful when the array is external and the actual definition, which allocates storage, is given elsewhere. The first constant expression may also be omitted when the declarator is followed by initialization. In this case the size is calculated from the number of initial elements supplied.

An array may be constructed from one of the basic types, from a pointer, from a structure or union, or from another array (to generate a multidimensional array).

Not all the possibilities allowed by the syntax above are actually permitted. The restrictions are as follows: functions may not return arrays or functions although they may return pointers; there are no arrays of functions although there may be arrays of pointers to functions. Likewise, a structure or union may not contain a function; but it may contain a pointer to a function.

As an example, the declaration

int i, ∗ip, f(), ∗fip(), (∗pfi)();

declares an integer **i**, a pointer **ip** to an integer, a function **f** returning an integer, a function **fip** returning a pointer to an integer, and a pointer **pfi** to a function which returns an integer. It is especially useful to compare the last two. The binding of ∗**fip()** is ∗(**fip()**). The declaration suggests, and the same construction in an expression requires, the calling of a function **fip**. Using indirection through the (pointer) result to yield an integer. In the declarator **(∗pfi)()**, the extra parentheses are necessary, as they are also in an expression, to indicate that

indirection through a pointer to a function yields a function, which is then called; it returns an integer.

As another example,

float fa[17], *afp[17];

declares an array of **float** numbers and an array of pointers to **float** numbers. Finally,

static int x3d[3][5][7];

declares a static 3-dimensional array of integers, with rank 3×5×7. In complete detail, **x3d** is an array of three items; each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions **x3d**, **x3d[i]**, **x3d[i][j]**, **x3d[i][j][k]** may reasonably appear in an expression. The first three have type "array" and the last has type **int**.

8.5. Structure and Union Declarations

A structure is an object consisting of a sequence of named members. Each member may have any type. A union is an object which may, at a given time, contain any one of several members. Structure and union specifiers have the same form.

struct-or-union-specifier:
struct-or-union { struct-decl-list }
struct-or-union identifier { struct-decl-list }
struct-or-union identifier

struct-or-union:
struct
union

The struct-decl-list is a sequence of declarations for the members of the structure or union:

struct-decl-list:
struct-declaration
struct-declaration struct-decl-list

struct-declaration:
type-specifier struct-declarator-list ;

struct-declarator-list:
struct-declarator
struct-declarator , struct-declarator-list

In the usual case, a struct-declarator is just a declarator for a member of a structure or union. A structure member may also consist of a specified number of bits. Such a member is also called a *field ;* its length, a non-negative constant expression, is set off from the field name by a colon.

struct-declarator:
declarator
declarator : constant-expression
: constant-expression

Within a structure, the objects declared have addresses which increase as the declarations are read left to right. Each nonfield member of a structure begins on an addressing boundary appropriate to its type; therefore, there may be unnamed holes in a structure. Field members are packed into machine integers; they do not straddle words. A field which does not fit into the space remaining in a word is put into the next word. No field may be wider than a word.

Fields are assigned right to left on the PDP-11 and VAX-11, left to right on the 3B 20.

A struct-declarator with no declarator, only a colon and a width, indicates an unnamed field useful for padding to conform to externally-imposed layouts. As a special case, a field with a width of 0 specifies alignment of the next field at an implementation dependant boundary.

The language does not restrict the types of things that are declared as fields, but implementations are not required to support any but integer fields. Moreover, even **int** fields may be considered to be unsigned. On the PDP-11, fields are not signed and have only integer values; on the VAX-11, fields declared with **int** are treated as containing a sign. For these reasons, it is strongly recommended that fields be declared as **unsigned**. In all implementations, there are no arrays of fields, and the address-of operator **&** may not be applied to them, so that there are no pointers to fields.

A union may be thought of as a structure all of whose members begin at offset 0 and whose size is sufficient to contain any of its members. At most, one of the members can be stored in a union at any time.

A structure or union specifier of the second form, that is, one of

struct *identifier { struct-decl-list }*
union *identifier { struct-decl-list }*

declares the identifier to be the *structure tag* (or union tag) of the structure specified by the list. A subsequent declaration may then use the third form of specifier, one of

struct *identifier*
union *identifier*

Structure tags allow definition of self-referential structures. Structure tags also permit the long part of the declaration to be given once and used several times. It is illegal to declare a structure or union which contains an instance of itself, but a structure or union may contain a pointer to an instance of itself.

The third form of a structure or union specifier may be used prior to a declaration which gives the complete specification of the structure or union in situations in which the size of the structure or union is unnecessary. The size is unnecessary in two situations: when a pointer to a structure or union is being declared and when a **typedef** name is declared to be a synonym for a structure or union. This, for example, allows the declaration of a pair of structures which contain pointers to each other.

The names of members and tags do not conflict with each other or with ordinary variables. A particular name may not be used twice in the same structure, but the same name may be used in several different structures in the same scope.

A simple but important example of a structure declaration is the following binary tree structure:

```
struct tnode
{
        char tword[20];
        int count;
        struct tnode *left;
        struct tnode *right;
};
```

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration has been given, the declaration

```
struct tnode s, *sp;
```

declares **s** to be a structure of the given sort and **sp** to be a pointer to a structure of the given sort. With these declarations, the expression

```
sp->count
```

refers to the **count** field of the structure to which **sp** points;

s.left

refers to the left subtree pointer of the structure **s**; and

s.right->tword[0]

refers to the first character of the **tword** member of the right subtree of **s**.

8.6. Enumeration Declarations

Enumeration variables and constants have integral type.

enum-specifier:
enum *{ enum-list }*
enum *identifier { enum-list }*
enum *identifier*

enum-list:
enumerator
enum-list , enumerator

enumerator:
identifier
identifier = constant-expression

The identifiers in an enum-list are declared as constants and may appear wherever constants are required. If no enumerators with **=** appear, then the values of the corresponding constants begin at 0 and increase by 1 as the declaration is read from left to right. An enumerator with **=** gives the associated identifier the value indicated; subsequent identifiers continue the progression from the assigned value.

The names of enumerators in the same scope must all be distinct from each other and from those of ordinary variables.

The role of the identifier in the enum-specifier is entirely analogous to that of the structure tag in a struct-specifier; it names a particular enumeration. For example,

```
enum color { chartreuse, burgundy, claret=20, winedark };
...
enum color **cp, col;
...
col = claret;
cp = &col;
...
if (**cp == burgundy) ...
```

makes **color** the enumeration-tag of a type describing various colors, and then declares **cp** as a pointer to an object of that type, and **col** as an object of that type. The possible values are drawn from the set {0,1,20,21}.

8.7. Initialization

A declarator may specify an initial value for the identifier being declared. The initializer is preceded by **=** and consists of an expression or a list of values nested in braces.

initializer:
= expression
= { initializer-list }
= { initializer-list , }

initializer-list:
expression
initializer-list , initializer-list
{ *initializer-list* }
{ *initializer-list* , }

All the expressions in an initializer for a static or external variable must be constant expressions, which are described in ''CONSTANT EXPRESSIONS'', or expressions which reduce to the address of a previously declared variable, possibly offset by a constant expression. Automatic or register variables may be initialized by arbitrary expressions involving constants and previously declared variables and functions.

Static and external variables that are not initialized are guaranteed to start off as zero. Automatic and register variables that are not initialized are guaranteed to start off as garbage.

When an initializer applies to a *scalar* (a pointer or an object of arithmetic type), it consists of a single expression, perhaps in braces. The initial value of the object is taken from the expression; the same conversions as for assignment are performed.

When the declared variable is an *aggregate* (a structure or array), the initializer consists of a brace-enclosed, comma-separated list of initializers for the members of the aggregate written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the aggregate. If there are fewer initializers in the list than there are members of the aggregate, then the aggregate is padded with zeros. It is not permitted to initialize unions or automatic aggregates.

Braces may in some cases be omitted. If the initializer begins with a left brace, then the succeeding comma-separated list of initializers initializes the members of the aggregate; it is erroneous for there to be more initializers than members. If, however, the initializer does not begin with a left brace, then only enough elements from the list are taken to account for the members of the aggregate; any remaining members are left to initialize the next member of the aggregate of which the current aggregate is a part.

A final abbreviation allows a **char** array to be initialized by a string. In this case successive characters of the string initialize the members of the array.

For example,

```
int x[] = { 1, 3, 5 };
```

declares and initializes **x** as a one-dimensional array which has three members, since no size was specified and there are three initializers.

```
float y[4][3] =
{
        { 1, 3, 5 },
        { 2, 4, 6 },
        { 3, 5, 7 },
};
```

is a completely-bracketed initialization: 1, 3, and 5 initialize the first row of the array **y[0]**, namely **y[0][0]**, **y[0][1]**, and **y[0][2]**. Likewise, the next two lines initialize **y[1]** and **y[2]**. The initializer ends early and therefore **y[3]** is initialized with 0. Precisely, the same effect could have been achieved by

```
float y[4][3] =
{
        1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for **y** begins with a left brace but that for **y[0]** does not; therefore, three elements from the

list are used. Likewise, the next three are taken successively for **y[1]** and **y[2]**. Also,

```
float y[4][3] =
{
        { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of **y** (regarded as a two-dimensional array) and leaves the rest 0.

Finally,

```
char msg[] = "Syntax error on line %s\n";
```

shows a character array whose members are initialized with a string.

8.8. Type Names

In two contexts (to specify type conversions explicitly by means of a cast and as an argument of **sizeof**), it is desired to supply the name of a data type. This is accomplished using a ''type name'', which in essence is a declaration for an object of that type which omits the name of the object.

type-name:
type-specifier abstract-declarator

abstract-declarator:
empty
(abstract-declarator)
** abstract-declarator*
abstract-declarator ()
abstract-declarator [*constant-expression*$_{opt}$]

To avoid ambiguity, in the construction

(abstract-declarator)

the abstract-declarator is required to be nonempty. Under this restriction, it is possible to identify uniquely the location in the abstract-declarator where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. For example,

int
int *
int ***[3]**
int (*)**[3]**
int *()
int (*)()
int (***[3]**)()

name respectively the types ''integer,'' ''pointer to integer,'' ''array of three pointers to integers,'' ''pointer to an array of three integers,'' ''function returning pointer to integer,'' ''pointer to function returning an integer,'' and ''array of three pointers to functions returning an integer.''

8.9. Typedef

Declarations whose ''storage class'' is **typedef** do not define storage but instead define identifiers which can be used later as if they were type keywords naming fundamental or derived types.

typedef-name:
identifier

Within the scope of a declaration involving **typedef**, each identifier appearing as part of any declarator therein becomes syntactically equivalent to the type keyword naming the type associated with the identifier in the

way described in ‘‘Meaning of Declarators.’’ For example, after

typedef int MILES, *KLICKSP;
typedef struct { double re, im; } complex;

the constructions

MILES distance;
extern KLICKSP metricp;
complex z, *zp;

are all legal declarations; the type of **distance** is **int**, that of **metricp** is ‘‘pointer to **int**, ’’ and that of **z** is the specified structure. The **zp** is a pointer to such a structure.

The **typedef** does not introduce brand-new types, only synonyms for types which could be specified in another way. Thus in the example above **distance** is considered to have exactly the same type as any other **int** object.

9. Statements

Except as indicated, statements are executed in sequence.

9.1. Expression Statement

Most statements are expression statements, which have the form

expression ;

Usually expression statements are assignments or function calls.

9.2. Compound Statement or Block

So that several statements can be used where one is expected, the compound statement (also, and equivalently, called ‘‘block’’) is provided:

compound-statement:
{ declaration-list$_{opt}$ *statement-list*$_{opt}$ *}*

declaration-list:
declaration
declaration declaration-list

statement-list:
statement
statement statement-list

If any of the identifiers in the declaration-list were previously declared, the outer declaration is pushed down for the duration of the block, after which it resumes its force.

Any initializations of **auto** or **register** variables are performed each time the block is entered at the top. It is currently possible (but a bad practice) to transfer into a block; in that case the initializations are not performed. Initializations of **static** variables are performed only once when the program begins execution. Inside a block, **extern** declarations do not reserve storage so initialization is not permitted.

9.3. Conditional Statement

The two forms of the conditional statement are

if (*expression*) *statement*
if (*expression*) *statement* **else** *statement*

In both cases, the expression is evaluated; and if it is nonzero, the first substatement is executed. In the second case, the second substatement is executed if the expression is 0. The ''else'' ambiguity is resolved by connecting an **else** with the last encountered **else**-less **if**.

9.4. While Statement

The **while** statement has the form

while (*expression*) *statement*

The substatement is executed repeatedly so long as the value of the expression remains nonzero. The test takes place before each execution of the statement.

9.5. Do Statement

The **do** statement has the form

do *statement* **while** (*expression*) ;

The substatement is executed repeatedly until the value of the expression becomes 0. The test takes place after each execution of the statement.

9.6. For Statement

The **for** statement has the form:

for *(exp-1*$_{opt}$ *; exp-2*$_{opt}$ *; exp-3*$_{opt}$ *) statement*

Except for the behavior of **continue**, this statement is equivalent to

```
exp-1 ;
while ( exp-2 )
{
        statement
        exp-3 ;
}
```

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0. The third expression often specifies an incrementing that is performed after each iteration.

Any or all of the expressions may be dropped. A missing *exp-2* makes the implied **while** clause equivalent to **while(1)**; other missing expressions are simply dropped from the expansion above.

9.7. Switch Statement

The **switch** statement causes control to be transferred to one of several statements depending on the value of an expression. It has the form

switch (*expression*) *statement*

The usual arithmetic conversion is performed on the expression, but the result must be **int**. The statement is typically compound. Any statement within the statement may be labeled with one or more case prefixes as follows:

case *constant-expression* :

where the constant expression must be **int**. No two of the case constants in the same switch may have the same value. Constant expressions are precisely defined in ''CONSTANT EXPRESSIONS.''

There may also be at most one statement prefix of the form

default :

When the **switch** statement is executed, its expression is evaluated and compared with each case constant. If one of the case constants is equal to the value of the expression, control is passed to the statement following the matched case prefix. If no case constant matches the expression and if there is a **default**, prefix, control passes to the prefixed statement. If no case matches and if there is no **default**, then none of the statements in the switch is executed.

The prefixes **case** and **default** do not alter the flow of control, which continues unimpeded across such prefixes. To exit from a switch, see ''Break Statement.''

Usually, the statement that is the subject of a switch is compound. Declarations may appear at the head of this statement, but initializations of automatic or register variables are ineffective.

9.8. Break Statement

The statement

break ;

causes termination of the smallest enclosing **while**, **do**, **for**, or **switch** statement; control passes to the statement following the terminated statement.

9.9. Continue Statement

The statement

continue ;

causes control to pass to the loop-continuation portion of the smallest enclosing **while**, **do**, or **for** statement; that is to the end of the loop. More precisely, in each of the statements

```
while (...) {                do {                    for (...) {
        statement ;                  statement ;             statement ;
        contin: ;                    contin: ;               contin: ;
    }                            } while (...);          }
```

a **continue** is equivalent to **goto contin**. (Following the **contin:** is a null statement, see ''Null Statement''.)

9.10. Return Statement

A function returns to its caller by means of the **return** statement which has one of the forms

return ;
return *expression* ;

In the first case, the returned value is undefined. In the second case, the value of the expression is returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value. The expression may be parenthesized.

9.11. Goto Statement

Control may be transferred unconditionally by means of the statement

goto *identifier* ;

The identifier must be a label (see ''Labeled Statement'') located in the current function.

9.12. Labeled Statement

Any statement may be preceded by label prefixes of the form

identifier :

which serve to declare the identifier as a label. The only use of a label is as a target of a **goto**. The scope of a label is the current function, excluding any subblocks in which the same identifier has been redeclared. See ''SCOPE RULES.''

9.13. Null Statement

The null statement has the form

;

A null statement is useful to carry a label just before the **}** of a compound statement or to supply a null body to a looping statement such as **while**.

10. External Definitions

A C program consists of a sequence of external definitions. An external definition declares an identifier to have storage class **extern** (by default) or perhaps **static**, and a specified type. The type-specifier (see ''Type Specifiers'' in ''DECLARATIONS'') may also be empty, in which case the type is taken to be **int**. The scope of external definitions persists to the end of the file in which they are declared just as the effect of declarations persists to the end of a block. The syntax of external definitions is the same as that of all declarations except that only at this level may the code for functions be given.

10.1. External Function Definitions

Function definitions have the form

function-definition:
decl-specifiers$_{opt}$ *function-declarator function-body*

The only sc-specifiers allowed among the decl-specifiers are **extern** or **static**; see ''Scope of Externals'' in ''SCOPE RULES'' for the distinction between them. A function declarator is similar to a declarator for a ''function returning ...'' except that it lists the formal parameters of the function being defined.

function-declarator:
declarator (parameter-list$_{opt}$ *)*

parameter-list:
identifier
identifier , parameter-list

The function-body has the form

function-body:
declaration-list$_{opt}$ *compound-statement*

The identifiers in the parameter list, and only those identifiers, may be declared in the declaration list. Any identifiers whose type is not given are taken to be **int**. The only storage class which may be specified is **register**; if it is specified, the corresponding actual parameter will be copied, if possible, into a register at the outset of the function.

A simple example of a complete function definition is

```
int max(a, b, c)
        int a, b, c;
{
        int m;

        m = (a > b) ? a : b;
        return((m > c) ? m : c);
}
```

Here **int** is the type-specifier; **max(a, b, c)** is the function-declarator; **int a, b, c;** is the declaration-list for the formal parameters; **{ ... }** is the block giving the code for the statement.

The C program converts all **float** actual parameters to **double**, so formal parameters declared **float** have their declaration adjusted to read **double**. All **char** and **short** formal parameter declarations are similarly adjusted to read **int**. Also, since a reference to an array in any context (in particular as an actual parameter) is taken to mean a pointer to the first element of the array, declarations of formal parameters declared "array of ..." are adjusted to read "pointer to ...."

10.2. External Data Definitions

An external data definition has the form

data-definition:
declaration

The storage class of such data may be **extern** (which is the default) or **static** but not **auto** or **register**.

11. Scope Rules

A C program need not all be compiled at the same time. The source text of the program may be kept in several files, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out both through explicit calls and through manipulation of external data.

Therefore, there are two kinds of scopes to consider: first, what may be called the lexical scope of an identifier, which is essentially the region of a program during which it may be used without drawing "undefined identifier" diagnostics; and second, the scope associated with external identifiers, which is characterized by the rule that references to the same external identifier are references to the same object.

11.1. Lexical Scope

The lexical scope of identifiers declared in external definitions persists from the definition through the end of the source file in which they appear. The lexical scope of identifiers which are formal parameters persists through the function with which they are associated. The lexical scope of identifiers declared at the head of a block persists until the end of the block. The lexical scope of labels is the whole of the function in which they appear.

In all cases, however, if an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of that identifier outside the block is suspended until the end of the block.

Remember also (see "Structure, Union, and Enumeration Declarations" in "DECLARATIONS") that tags, identifiers associated with ordinary variables, and identities associated with structure and union members form three disjoint classes which do not conflict. Members and tags follow the same scope rules as other identifiers. The **enum** constants are in the same class as ordinary variables and follow the same scope rules. The **typedef** names are in the same class as ordinary identifiers. They may be redeclared in inner blocks, but an explicit type must be given in the inner declaration:

```
typedef float distance;
...
{
    auto int distance;
    ...
}
```

The **int** must be present in the second declaration, or it would be taken to be a declaration with no declarators and type **distance**.

11.2. Scope of Externals

If a function refers to an identifier declared to be **extern**, then somewhere among the files or libraries constituting the complete program there must be at least one external definition for the identifier. All functions in a given program which refer to the same external identifier refer to the same object, so care must be taken that the type and size specified in the definition are compatible with those specified by each function which references the data.

It is illegal to explicitly initialize any external identifier more than once in the set of files and libraries comprising a multi-file program. It is legal to have more than one data definition for any external non-function identifier; explicit use of **extern** does not change the meaning of an external declaration.

In restricted environments, the use of the **extern** storage class takes on an additional meaning. In these environments, the explicit appearance of the **extern** keyword in external data declarations of identities without initialization indicates that the storage for the identifiers is allocated elsewhere, either in this file or another file. It is required that there be exactly one definition of each external identifier (without **extern**) in the set of files and libraries comprising a mult-file program.

Identifiers declared **static** at the top level in external definitions are not visible in other files. Functions may be declared **static**.

12. Compiler Control Lines

The C compiler contains a preprocessor capable of macro substitution, conditional compilation, and inclusion of named files. Lines beginning with **#** communicate with this preprocessor. There may be any number of blanks and horizontal tabs between the **#** and the directive. These lines have syntax independent of the rest of the language; they may appear anywhere and have effect which lasts (independent of scope) until the end of the source program file.

12.1. Token Replacement

A compiler-control line of the form

#define *identifier token-string*$_{opt}$

causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens. Semicolons in or at the end of the token-string are part of that string. A line of the form

#define *identifier(identifier, ...)token-string*$_{opt}$

where there is no space between the first identifier and the **(**, is a macro definition with arguments. There may be zero or more formal parameters. Subsequent instances of the first identifier followed by a **(**, a sequence of tokens delimited by commas, and a **)** are replaced by the token string in the definition. Each occurrence of an identifier mentioned in the formal parameter list of the definition is replaced by the corresponding token string from the call. The actual arguments in the call are token strings separated by commas; however, commas in quoted strings or protected by parentheses do not separate arguments. The number of formal and actual parameters must be the same. Strings and character constants in the token-string are scanned for formal parameters, but strings and character constants in the rest of the program are not scanned for defined identifiers to replacement.

In both forms the replacement string is rescanned for more defined identifiers. In both forms a long definition may be continued on another line by writing \ at the end of the line to be continued.

This facility is most valuable for definition of ‘‘manifest constants,’’ as in

```
#define TABSIZE 100

int table[TABSIZE];
```

A control line of the form

#undef *identifier*

causes the identifier's preprocessor definition (if any) to be forgotten.

If a **#define**d identifier is the subject of a subsequent **#define** with no intervening **#undef**, then the two token-strings are compared textually. If the two token-strings are not identical (all white space is considered as equivalent), then the identifier is considered to be redefined.

12.2. File Inclusion

A compiler control line of the form

#include *"filename"*

causes the replacement of that line by the entire contents of the file *filename*. The named file is searched for first in the directory of the file containing the **#include**, and then in a sequence of specified or standard places. Alternatively, a control line of the form

#include <*filename*>

searches only the specified or standard places and not the directory of the **#include**. (How the places are specified is not part of the language.)

#includes may be nested.

12.3. Conditional Compilation

A compiler control line of the form

#if *restricted-constant-expression*

checks whether the restricted-constant expression evaluates to nonzero. (Constant expressions are discussed in ‘‘CONSTANT EXPRESSIONS’’; the following additional restrictions apply here: the constant expression may not contain **sizeof** casts, or an enumeration constant.)

A restricted constant expression may also contain the additional unary expression

defined *identifier*

or

defined(*identifier*)

which evaluates to one if the identifier is currently defined in the preprocessor and zero if it is not.

All currently defined identifiers in restricted-constant-expressions are replaced by their token-strings (except those identifiers modified by **defined**) just as in normal text. The restricted constant expression will be evaluated only after all expressions have finished. During this evaluation, all undefined (to the procedure) identifiers evaluate to zero.

A control line of the form

#ifdef *identifier*

checks whether the identifier is currently defined in the preprocessor; i.e., whether it has been the subject of a **#define** control line. It is equivalent to **#ifdef(***identifier***)**. A control line of the form

#ifndef *identifier*

checks whether the identifier is currently undefined in the preprocessor. It is equivalent to

#if **!defined(***identifier***)**.

All three forms are followed by an arbitrary number of lines, possibly containing a control line

#else

and then by a control line

#endif

If the checked condition is true, then any lines between **#else** and **#endif** are ignored. If the checked condition is false, then any lines between the test and a **#else** or, lacking a **#else**, the **#endif** are ignored.

These constructions may be nested.

12.4. Line Control

For the benefit of other preprocessors which generate C programs, a line of the form

#line *constant "filename*"

causes the compiler to believe, for purposes of error diagnostics, that the line number of the next source line is given by the constant and the current input file is named by "*filename*". If "*filename*" is absent, the remembered file name does not change.

13. Implicit Declarations

It is not always necessary to specify both the storage class and the type of identifiers in a declaration. The storage class is supplied by the context in external definitions and in declarations of formal parameters and structure members. In a declaration inside a function, if a storage class but no type is given, the identifier is assumed to be **int**; if a type but no storage class is indicated, the identifier is assumed to be **auto**. An exception to the latter rule is made for functions because **auto** functions do not exist. If the type of an identifier is ‘‘function returning . . . ,’’ it is implicitly declared to be **extern**.

In an expression, an identifier followed by **(** and not already declared is contextually declared to be ‘‘function returning **int**.’’

14. Types Revisited

This part summarizes the operations which can be performed on objects of certain types.

14.1. Structures and Unions

Structures and unions may be assigned, passed as arguments to functions, and returned by functions. Other plausible operators, such as equality comparison and structure casts, are not implemented.

In a reference to a structure or union member, the name on the right of the **->** or the **.** must specify a member of the aggregate named or pointed to by the expression on the left. In general, a member of a union may not be inspected unless the value of the union has been assigned using that same member. However, one special guarantee is made by the language in order to simplify the use of unions: if a union contains several structures that share a common initial sequence and if the union currently contains one of these structures, it is permitted to inspect the common initial part of any of the contained structures. For example, the following is a legal fragment:

```
union
{
        struct
        {
                int     type;
        } n;
        struct
        {
                int     type;
                int     intnode;
        } ni;
        struct
        {
                int     type;
                float   floatnode;
        } nf;
} u;

...
u.nf.type = FLOAT;
u.nf.floatnode = 3.14;
...
if (u.n.type == FLOAT)
        ... sin(u.nf.floatnode) ...
```

14.2. Functions

There are only two things that can be done with a function **m**, call it or take its address. If the name of a function appears in an expression not in the function-name position of a call, a pointer to the function is generated. Thus, to pass one function to another, one might say

```
int f();
...
g(f);
```

Then the definition of **g** might read

```
g(funcp)
        int (*funcp)();
{
        ...
        (*funcp)();
        ...
}
```

Notice that **f** must be declared explicitly in the calling routine since its appearance in **g(f)** was not followed by **(**.

14.3. Arrays, Pointers, and Subscripting

Every time an identifier of array type appears in an expression, it is converted into a pointer to the first member of the array. Because of this conversion, arrays are not lvalues. By definition, the subscript operator **[]** is interpreted in such a way that **E1[E2]** is identical to ***((E1)+E2)**. Because of the conversion rules which apply to +, if **E1** is an array and **E2** an integer, then **E1[E2]** refers to the **E2-th** member of **E1**. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

A consistent rule is followed in the case of multidimensional arrays. If **E** is an *n*-dimensional array of rank i×j×...×k, then **E** appearing in an expression is converted to a pointer to an (n-1)-dimensional array with rank j×...×k. If the * operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to (n-1)-dimensional array, which itself is immediately converted into a pointer.

For example, consider

```
int x[3][5];
```

Here **x** is a 3×5 array of integers. When **x** appears in an expression, it is converted to a pointer to (the first of three) 5-membered arrays of integers. In the expression **x[i]**, which is equivalent to ***(x+i)**, **x** is first converted to a pointer as described; then **i** is converted to the type of **x**, which involves multiplying **i** by the length the object to which the pointer points, namely 5-integer objects. The results are added and indirection applied to yield an array (of five integers) which in turn is converted to a pointer to the first of the integers. If there is another subscript, the same argument applies again; this time the result is an integer.

Arrays in C are stored row-wise (last subscript varies fastest) and the first subscript in the declaration helps determine the amount of storage consumed by an array. Arrays play no other part in subscript calculations.

14.4. Explicit Pointer Conversions

Certain conversions involving pointers are permitted but have implementation-dependent aspects. They are all specified by means of an explicit type-conversion operator, see ''Unary Operators'' under''EXPRESSIONS'' and ''Type Names''under ''DECLARATIONS.''

A pointer may be converted to any of the integral types large enough to hold it. Whether an **int** or **long** is required is machine dependent. The mapping function is also machine dependent but is intended to be unsurprising to those who know the addressing structure of the machine. Details for some particular machines are given below.

An object of integral type may be explicitly converted to a pointer. The mapping always carries an integer converted from a pointer back to the same pointer but is otherwise machine dependent.

A pointer to one type may be converted to a pointer to another type. The resulting pointer may cause addressing exceptions upon use if the subject pointer does not refer to an object suitably aligned in storage. It is guaranteed that a pointer to an object of a given size may be converted to a pointer to an object of a smaller size and back again without change.

For example, a storage-allocation routine might accept a size (in bytes) of an object to allocate, and return a **char** pointer; it might be used in this way.

```
extern char *malloc();
double *dp;

dp = (double *) malloc(sizeof(double));
*dp = 22.0 / 7.0;
```

The **alloc** must ensure (in a machine-dependent way) that its return value is suitable for conversion to a pointer to **double**; then the *use* of the function is portable.

The pointer representation on the PDP-11 corresponds to a 16-bit integer and measures bytes. The **char**'s have no alignment requirements; everything else must have an even address.

On the VAX-11, pointers are 32 bits long and measure bytes. Elementary objects are aligned on a boundary equal to their length, except that **double** quantities need be aligned only on even 4-byte boundaries. Aggregates are aligned on the strictest boundary required by any of their constituents.

The 3B 20 computer has 24-bit pointers placed into 32-bit quantities. Most objects are aligned on 4-byte boundaries. **Short**s are aligned in all cases on 2-byte boundaries. Arrays of characters, all structures, **int**s, **long**s, **float**s, and **double**s are aligned on 4-byte boundries; but structure members may be packed tighter.

14.5. CONSTANT EXPRESSIONS

In several places C requires expressions that evaluate to a constant: after **case**, as array bounds, and in initializers. In the first two cases, the expression can involve only integer constants, character constants, casts to integral types, enumeration constants, and **sizeof** expressions, possibly connected by the binary operators

+ − * / % & | ^ << >> == != < > <= >= && ||

or by the unary operators

− ~

or by the ternary operator

?:

Parentheses can be used for grouping but not for function calls.

More latitude is permitted for initializers; besides constant expressions as discussed above, one can also use floating constants and arbitrary casts and can also apply the unary **&** operator to external or static objects and to external or static arrays subscripted with a constant expression. The unary **&** can also be applied implicitly by appearance of unsubscripted arrays and functions. The basic rule is that initializers must evaluate either to a constant or to the address of a previously declared external or static object plus or minus a constant.

15. Portability Considerations

Certain parts of C are inherently machine dependent. The following list of potential trouble spots is not meant to be all-inclusive but to point out the main ones.

Purely hardware issues like word size and the properties of floating point arithmetic and integer division have proven in practice to be not much of a problem. Other facets of the hardware are reflected in differing implementations. Some of these, particularly sign extension (converting a negative character into a negative integer) and the order in which bytes are placed in a word, are nuisances that must be carefully watched. Most of the others are only minor problems.

The number of **register** variables that can actually be placed in registers varies from machine to machine as does the set of valid types. Nonetheless, the compilers all do things properly for their own machine; excess or invalid **register** declarations are ignored.

Some difficulties arise only when dubious coding practices are used. It is exceedingly unwise to write programs that depend on any of these properties.

The order of evaluation of function arguments is not specified by the language. The order in which side effects take place is also unspecified.

Since character constants are really objects of type **int**, multicharacter character constants may be permitted. The specific implementation is very machine dependent because the order in which characters are assigned to a word varies from one machine to another.

Fields are assigned to words and characters to integers right to left on some machines and left to right on other machines. These differences are invisible to isolated programs that do not indulge in type punning (e.g., by converting an **int** pointer to a **char** pointer and inspecting the pointed-to storage) but must be accounted for when conforming to externally-imposed storage layouts.

16. Syntax Summary

This summary of C syntax is intended more for aiding comprehension than as an exact statement of the

language.

16.1. Expressions

The basic expressions are:

expression:
primary
** expression*
&lvalue
− expression
! expression
~ expression
++ lvalue
−−lvalue
lvalue ++
lvalue −−
sizeof *expression*
sizeof (*type-name***)**
(type-name) expression
expression binop expression
expression ? expression : expression
lvalue asgnop expression
expression , expression

primary:
identifier
constant
string
(expression)
primary (expression-list$_{opt}$ *)*
primary [expression]
primary . identifier
primary − identifier

lvalue:
identifier
primary [expression]
lvalue . identifier
primary − identifier
** expression*
(lvalue)

The primary-expression operators

() [] . −

have highest priority and group left to right. The unary operators

* & − ! ~ ++ −− **sizeof** *(type-name*)

have priority below the primary operators but higher than any binary operator and group right to left. Binary operators group left to right; they have priority decreasing as indicated below.

binop:
* / %
+ −
>> <<
< > <= >=
== !=
&
^
|
&&
||

The conditional operator groups right to left.

Assignment operators all have the same priority and all group right to left.

asgnop:
= += −= *= /= %= >>= <<= &= ^= |=

The comma operator has the lowest priority and groups left to right.

16.2. Declarations

declaration:
decl-specifiers init-declarator-list$_{opt}$;

decl-specifiers:
type-specifier decl-specifiers$_{opt}$
sc-specifier decl-specifiers$_{opt}$

sc-specifier:
auto
static
extern
register
typedef

type-specifier:
struct-or-union-specifier
typedef-name
enum-specifier
basic-type-specifier:
basic-type
basic-type basic-type-specifiers
basic-type:
char
short
int
long
unsigned
float
double
void

enum-specifier:
enum *{ enum-list }*
enum *identifier { enum-list }*
enum *identifier*

enum-list:
enumerator
enum-list , enumerator

enumerator:
identifier
identifier = constant-expression

init-declarator-list:
init-declarator
init-declarator , init-declarator-list

init-declarator:
declarator initializer$_{opt}$

declarator:
identifier
(declarator)
** declarator*
declarator ()
declarator [constant-expression$_{opt}$ *]*

struct-or-union-specifier:
struct *{ struct-decl-list }*
struct *identifier { struct-decl-list }*
struct *identifier*
union { *struct-decl-list }*
union *identifier { struct-decl-list }*
union *identifier*

struct-decl-list:
struct-declaration
struct-declaration struct-decl-list

struct-declaration:
type-specifier struct-declarator-list ;

struct-declarator-list:
struct-declarator
struct-declarator , struct-declarator-list

struct-declarator:
declarator
declarator : constant-expression
: constant-expression

initializer:
= expression
= { initializer-list }
= { initializer-list , }

initializer-list:
expression
initializer-list , initializer-list
{ initializer-list }
{ initializer-list , }

type-name:
type-specifier abstract-declarator

abstract-declarator:
empty
(abstract-declarator)
** abstract-declarator*
abstract-declarator ()
abstract-declarator [constant-expression$_{opt}$ *]*

typedef-name:
identifier

16.3. Statements

compound-statement:
{ declaration-list$_{opt}$ *statement-list*$_{opt}$ *}*

declaration-list:
declaration
declaration declaration-list

statement-list:
statement
statement statement-list

statement:
compound-statement
expression ;
if *(expression) statement*
if *(expression) statement* **else** *statement*
while *(expression) statement*
do *statement* **while** *(expression) ;*
for *(*exp_{opt}**;**exp_{opt}**;**exp_{opt}*) statement*
switch *(expression) statement*
case *constant-expression : statement*
default *: statement*
break ;
continue ;
return ;
return *expression ;*
goto *identifier ;*
identifier : statement
;

16.4. External definitions

program:
external-definition
external-definition program

external-definition:
function-definition
data-definition

function-definition:
decl-specifier$_{opt}$ *function-declarator function-body*

function-declarator:
declarator (parameter-list$_{opt}$ *)*

parameter-list:
identifier
identifier , parameter-list

function-body:
declaration-list$_{opt}$ *compound-statement*

data-definition:
extern *declaration* **;**
static *declaration* **;**

17. Preprocessor

#define *identifier token-string*$_{opt}$
#define *identifier*(*identifier*,...)*token-string*$_{opt}$
#undef *identifier*
#include "*filename*"
#include <*filename*>
#if *restricted-constant-expression*
#ifdef *identifier*
#ifndef *identifier*
#else
#endif
#line *constant* "*filename*"

Introduction
1. Sources of information
1.1. Text editing documents
1.2. Documentation describing UNIX
1.3. Pascal documents
1.4. Pascal User Manual
2. Basic UNIX Pascal
2.1. A first program
2.2. Larger programs
2.3. Correcting the first errors
2.4. Execution time errors
2.5. Features of the Pascal implementation
2.6. Interfacing to other languages
3. Using the Pascal interpreter
3.1. Introduction
3.2. Error diagnostics
3.3. Run time errors
3.4. Post-mortem debugging
3.5. Interactive debugging
3.6. Profiling
3.7. Tracing
3.8. Assertions
3.9. Arguments
3.10. Standard files
3.11. Separate compilation
3.12. Pitfalls
4. Implementation limits
5. Implementation notes
5.1. Portability
5.2. Pascal vs. UNIX standard I/O
5.3. Compile speed
5.4. Object code sizes
5.5. Run time format
5.6. Source file format
5.7. Additional predefined procedures and functions
5.8. Additional predefined types
5.9. Additional predefined variables
5.10. Extensions
5.11. Restrictions
5.12. Efficiency
5.13. Compatibility
6. Acknowledgements
7. References
Appendix A. Grammar
A.1. Lexical grammar
A.2. Syntactic grammar
Appendix B. Standard prelude
Appendix C. Differences from Berkeley Pascal 3.0
Appendix D. Differences from Berkeley Pascal 3.1
Appendix E. Differences from Berkeley Pascal 4.0
Appendix F. Differences from Berkeley Pascal 4.1
Appendix G. Differences from Berkeley Pascal 4.2
Appendix H. Differences from Berkeley Pascal 5.0
Appendix I. Differences from Berkeley Pascal 5.1
Appendix J. Differences from Berkeley Pascal 5.2
Appendix K. Differences from Berkeley Pascal 6.0
Appendix L. Differences from Berkeley Pascal 6.1
Appendix M. Differences from Berkeley Pascal 6.2
Appendix N. Differences from Berkeley Pascal 7.0
Appendix O. Differences from Berkeley Pascal 7.1
Appendix P. Differences from Berkeley Pascal 7.2
Appendix Q. Differences from Berkeley Pascal 8.0
Appendix R. Differences from Berkeley Pascal 8.1
Appendix S. Differences from Berkeley Pascal 8.2
Appendix T. Differences from Berkeley Pascal 8.3
Appendix U. Differences from Berkeley Pascal 8.4
Appendix V. Differences from Berkeley Pascal 8.5
Appendix W. Differences from Berkeley Pascal 8.6
Appendix X. Differences from Berkeley Pascal 8.7
Appendix Y. Differences from Berkeley Pascal 8.8
Appendix Z. Differences from Berkeley Pascal 8.9
Appendix AA. Differences from Berkeley Pascal 8.10
Appendix BB. Differences from Berkeley Pascal 8.11
Appendix CC. Differences from Berkeley Pascal 8.12
Appendix DD. Differences from Berkeley Pascal 8.13
Appendix EE. Differences from Berkeley Pascal 8.14
Appendix FF. Differences from Berkeley Pascal 8.15
Appendix GG. Differences from Berkeley Pascal 8.16
Appendix HH. Differences from Berkeley Pascal 8.17
Appendix II. Differences from Berkeley Pascal 8.18
Appendix JJ. Differences from Berkeley Pascal 8.19
Appendix KK. Differences from Berkeley Pascal 8.20
Appendix LL. Differences from Berkeley Pascal 8.21
Appendix MM. Differences from Berkeley Pascal 8.22
Appendix NN. Differences from Berkeley Pascal 8.23
Appendix OO. Differences from Berkeley Pascal 8.24
Appendix PP. Differences from Berkeley Pascal 8.25
Appendix QQ. Differences from Berkeley Pascal 8.26
Appendix RR. Differences from Berkeley Pascal 8.27
Appendix SS. Differences from Berkeley Pascal 8.28
Appendix TT. Differences from Berkeley Pascal 8.29
Appendix UU. Differences from Berkeley Pascal 8.30
Appendix VV. Differences from Berkeley Pascal 8.31
Appendix WW. Differences from Berkeley Pascal 8.32
Appendix XX. Differences from Berkeley Pascal 8.33
Appendix YY. Differences from Berkeley Pascal 8.34
Appendix ZZ. Differences from Berkeley Pascal 8.35
Appendix AAA. Differences from Berkeley Pascal 8.36
Appendix BBB. Differences from Berkeley Pascal 8.37
Appendix CCC. Differences from Berkeley Pascal 8.38
Appendix DDD. Differences from Berkeley Pascal 8.39
Appendix EEE. Differences from Berkeley Pascal 8.40
Appendix FFF. Differences from Berkeley Pascal 8.41
Appendix GGG. Differences from Berkeley Pascal 8.42
Appendix HHH. Differences from Berkeley Pascal 8.43
Appendix III. Differences from Berkeley Pascal 8.44
Appendix JJJ. Differences from Berkeley Pascal 8.45
Appendix KKK. Differences from Berkeley Pascal 8.46
Appendix LLL. Differences from Berkeley Pascal 8.47
Appendix MMM. Differences from Berkeley Pascal 8.48
Appendix NNN. Differences from Berkeley Pascal 8.49
Appendix OOO. Differences from Berkeley Pascal 8.50
Appendix PPP. Differences from Berkeley Pascal 8.51
Appendix QQQ. Differences from Berkeley Pascal 8.52
Appendix RRR. Differences from Berkeley Pascal 8.53
Appendix SSS. Differences from Berkeley Pascal 8.54
Appendix TTT. Differences from Berkeley Pascal 8.55
Appendix UUU. Differences from Berkeley Pascal 8.56
Appendix VVV. Differences from Berkeley Pascal 8.57
Appendix WWW. Differences from Berkeley Pascal 8.58
Appendix XXX. Differences from Berkeley Pascal 8.59
Appendix YYY. Differences from Berkeley Pascal 8.60
Appendix ZZZ. Differences from Berkeley Pascal 8.61
Appendix AAAA. Differences from Berkeley Pascal 8.62
Appendix BBBB. Differences from Berkeley Pascal 8.63
Appendix CCCC. Differences from Berkeley Pascal 8.64
Appendix DDDD. Differences from Berkeley Pascal 8.65
Appendix EEEE. Differences from Berkeley Pascal 8.66
Appendix FFFF. Differences from Berkeley Pascal 8.67
Appendix GGGG. Differences from Berkeley Pascal 8.68
Appendix HHHH. Differences from Berkeley Pascal 8.69
Appendix IIII. Differences from Berkeley Pascal 8.70
Appendix JJJJ. Differences from Berkeley Pascal 8.71
Appendix KKKK. Differences from Berkeley Pascal 8.72
Appendix LLLL. Differences from Berkeley Pascal 8.73
Appendix MMMM. Differences from Berkeley Pascal 8.74
Appendix NNNN. Differences from Berkeley Pascal 8.75
Appendix OOOO. Differences from Berkeley Pascal 8.76
Appendix PPPP. Differences from Berkeley Pascal 8.77
Appendix QQQQ. Differences from Berkeley Pascal 8.78
Appendix RRRR. Differences from Berkeley Pascal 8.79
Appendix SSSS. Differences from Berkeley Pascal 8.80
Appendix TTTT. Differences from Berkeley Pascal 8.81
Appendix UUUU. Differences from Berkeley Pascal 8.82
Appendix VVVV. Differences from Berkeley Pascal 8.83
Appendix WWWW. Differences from Berkeley Pascal 8.84
Appendix XXXX. Differences from Berkeley Pascal 8.85
Appendix YYYY. Differences from Berkeley Pascal 8.86
Appendix ZZZZ. Differences from Berkeley Pascal 8.87
Appendix AAAAA. Differences from Berkeley Pascal 8.88
Appendix BBBBB. Differences from Berkeley Pascal 8.89
Appendix CCCCC. Differences from Berkeley Pascal 8.90
Appendix DDDDD. Differences from Berkeley Pascal 8.91
Appendix EEEEE. Differences from Berkeley Pascal 8.92
Appendix FFFFF. Differences from Berkeley Pascal 8.93
Appendix GGGGG. Differences from Berkeley Pascal 8.94
Appendix HHHHH. Differences from Berkeley Pascal 8.95
Appendix IIIII. Differences from Berkeley Pascal 8.96
Appendix JJJJJ. Differences from Berkeley Pascal 8.97
Appendix KKKKK. Differences from Berkeley Pascal 8.98
Appendix LLLLL. Differences from Berkeley Pascal 8.99
Appendix MMMMM. Differences from Berkeley Pascal 8.100
Appendix NNNNN. Differences from Berkeley Pascal 8.101
Appendix OOOOO. Differences from Berkeley Pascal 8.102
Appendix PPPPP. Differences from Berkeley Pascal 8.103
Appendix QQQQQ. Differences from Berkeley Pascal 8.104
Appendix RRRRR. Differences from Berkeley Pascal 8.105
Appendix SSSSS. Differences from Berkeley Pascal 8.106
Appendix TTTTT. Differences from Berkeley Pascal 8.107
Appendix UUUUU. Differences from Berkeley Pascal 8.108
Appendix VVVVV. Differences from Berkeley Pascal 8.109
Appendix WWWWW. Differences from Berkeley Pascal 8.110
Appendix XXXXX. Differences from Berkeley Pascal 8.111
Appendix YYYYY. Differences from Berkeley Pascal 8.112
Appendix ZZZZZ. Differences from Berkeley Pascal 8.113
Appendix AAAAAA. Differences from Berkeley Pascal 8.114
Appendix BBBBBB. Differences from Berkeley Pascal 8.115
Appendix CCCCCC. Differences from Berkeley Pascal 8.116
Appendix DDDDDD. Differences from Berkeley Pascal 8.117
Appendix EEEEEE. Differences from Berkeley Pascal 8.118
Appendix FFFFFF. Differences from Berkeley Pascal 8.119
Appendix GGGGGG. Differences from Berkeley Pascal 8.120
Appendix HHHHHH. Differences from Berkeley Pascal 8.121
Appendix IIIIII. Differences from Berkeley Pascal 8.122
Appendix JJJJJJ. Differences from Berkeley Pascal 8.123
Appendix KKKKKK. Differences from Berkeley Pascal 8.124
Appendix LLLLLL. Differences from Berkeley Pascal 8.125
Appendix MMMMMM. Differences from Berkeley Pascal 8.126
Appendix NNNNNN. Differences from Berkeley Pascal 8.127
Appendix OOOOOO. Differences from Berkeley Pascal 8.128
Appendix PPPPPP. Differences from Berkeley Pascal 8.129
Appendix QQQQQQ. Differences from Berkeley Pascal 8.130
Appendix RRRRRR. Differences from Berkeley Pascal 8.131
Appendix SSSSSS. Differences from Berkeley Pascal 8.132
Appendix TTTTTT. Differences from Berkeley Pascal 8.133
Appendix UUUUUU. Differences from Berkeley Pascal 8.134
Appendix VVVVVV. Differences from Berkeley Pascal 8.135
Appendix WWWWWW. Differences from Berkeley Pascal 8.136
Appendix XXXXXX. Differences from Berkeley Pascal 8.137
Appendix YYYYYY. Differences from Berkeley Pascal 8.138
Appendix ZZZZZZ. Differences from Berkeley Pascal 8.139
Appendix AAAAAAA. Differences from Berkeley Pascal 8.140
Appendix BBBBBBB. Differences from Berkeley Pascal 8.141
Appendix CCCCCCC. Differences from Berkeley Pascal 8.142
Appendix DDDDDDD. Differences from Berkeley Pascal 8.143
Appendix EEEEEEE. Differences from Berkeley Pascal 8.144
Appendix FFFFFFF. Differences from Berkeley Pascal 8.145
Appendix GGGGGGG. Differences from Berkeley Pascal 8.146
Appendix HHHHHHH. Differences from Berkeley Pascal 8.147
Appendix IIIIIII. Differences from Berkeley Pascal 8.148
Appendix JJJJJJJ. Differences from Berkeley Pascal 8.149
Appendix KKKKKKK. Differences from Berkeley Pascal 8.150
Appendix LLLLLLL. Differences from Berkeley Pascal 8.151
Appendix MMMMMMM. Differences from Berkeley Pascal 8.152
Appendix NNNNNNN. Differences from Berkeley Pascal 8.153
Appendix OOOOOOO. Differences from Berkeley Pascal 8.154
Appendix PPPPPPP. Differences from Berkeley Pascal 8.155
Appendix QQQQQQQ. Differences from Berkeley Pascal 8.156
Appendix RRRRRRR. Differences from Berkeley Pascal 8.157
Appendix SSSSSSS. Differences from Berkeley Pascal 8.158
Appendix TTTTTTT. Differences from Berkeley Pascal 8.159
Appendix UUUUUUU. Differences from Berkeley Pascal 8.160
Appendix VVVVVVV. Differences from Berkeley Pascal 8.161
Appendix WWWWWWW. Differences from Berkeley Pascal 8.162
Appendix XXXXXXX. Differences from Berkeley Pascal 8.163
Appendix YYYYYYY. Differences from Berkeley Pascal 8.164
Appendix ZZZZZZZ. Differences from Berkeley Pascal 8.165
Appendix AAAAAAAA. Differences from Berkeley Pascal 8.166
Appendix BBBBBBBB. Differences from Berkeley Pascal 8.167
Appendix CCCCCCCC. Differences from Berkeley Pascal 8.168
Appendix DDDDDDDD. Differences from Berkeley Pascal 8.169
Appendix EEEEEEEE. Differences from Berkeley Pascal 8.170
Appendix FFFFFFFF. Differences from Berkeley Pascal 8.171
Appendix GGGGGGGG. Differences from Berkeley Pascal 8.172
Appendix HHHHHHHH. Differences from Berkeley Pascal 8.173
Appendix IIIIIIII. Differences from Berkeley Pascal 8.174
Appendix JJJJJJJJ. Differences from Berkeley Pascal 8.175
Appendix KKKKKKKK. Differences from Berkeley Pascal 8.176
Appendix LLLLLLLL. Differences from Berkeley Pascal 8.177
Appendix MMMMMMMM. Differences from Berkeley Pascal 8.178
Appendix NNNNNNNN. Differences from Berkeley Pascal 8.179
Appendix OOOOOOOO. Differences from Berkeley Pascal 8.180
Appendix PPPPPPPP. Differences from Berkeley Pascal 8.181
Appendix QQQQQQQQ. Differences from Berkeley Pascal 8.182
Appendix RRRRRRRR. Differences from Berkeley Pascal 8.183
Appendix SSSSSSSS. Differences from Berkeley Pascal 8.184
Appendix TTTTTTTT. Differences from Berkeley Pascal 8.185
Appendix UUUUUUUU. Differences from Berkeley Pascal 8.186
Appendix VVVVVVVV. Differences from Berkeley Pascal 8.187
Appendix WWWWWWWW. Differences from Berkeley Pascal 8.188
Appendix XXXXXXXX. Differences from Berkeley Pascal 8.189
Appendix YYYYYYYY. Differences from Berkeley Pascal 8.190
Appendix ZZZZZZZZ. Differences from Berkeley Pascal 8.191
Appendix AAAAAAAAA. Differences from Berkeley Pascal 8.192
Appendix BBBBBBBBB. Differences from Berkeley Pascal 8.193
Appendix CCCCCCCCC. Differences from Berkeley Pascal 8.194
Appendix DDDDDDDDD. Differences from Berkeley Pascal 8.195
Appendix EEEEEEEEE. Differences from Berkeley Pascal 8.196
Appendix FFFFFFFFF. Differences from Berkeley Pascal 8.197
Appendix GGGGGGGGG. Differences from Berkeley Pascal 8.198
Appendix HHHHHHHHH. Differences from Berkeley Pascal 8.199
Appendix IIIIIIIII. Differences from Berkeley Pascal 8.200
Appendix JJJJJJJJJ. Differences from Berkeley Pascal 8.201
Appendix KKKKKKKKK. Differences from Berkeley Pascal 8.202
Appendix LLLLLLLLL. Differences from Berkeley Pascal 8.203
Appendix MMMMMMMMM. Differences from Berkeley Pascal 8.204
Appendix NNNNNNNNN. Differences from Berkeley Pascal 8.205
Appendix OOOOOOOOO. Differences from Berkeley Pascal 8.206
Appendix PPPPPPPPP. Differences from Berkeley Pascal 8.207
Appendix QQQQQQQQQ. Differences from Berkeley Pascal 8.208
Appendix RRRRRRRRR. Differences from Berkeley Pascal 8.209
Appendix SSSSSSSSS. Differences from Berkeley Pascal 8.210
Appendix TTTTTTTTT. Differences from Berkeley Pascal 8.211
Appendix UUUUUUUUU. Differences from Berkeley Pascal 8.212
Appendix VVVVVVVVV. Differences from Berkeley Pascal 8.213
Appendix WWWWWWWWW. Differences from Berkeley Pascal 8.214
Appendix XXXXXXXXX. Differences from Berkeley Pascal 8.215
Appendix YYYYYYYYY. Differences from Berkeley Pascal 8.216
Appendix ZZZZZZZZZ. Differences from Berkeley Pascal 8.217
Appendix AAAAAAAAAA. Differences from Berkeley Pascal 8.218
Appendix BBBBBBBBBB. Differences from Berkeley Pascal 8.219
Appendix CCCCCCCCCC. Differences from Berkeley Pascal 8.220
Appendix DDDDDDDDDD. Differences from Berkeley Pascal 8.221
Appendix EEEEEEEEEE. Differences from Berkeley Pascal 8.222
Appendix FFFFFFFFFF. Differences from Berkeley Pascal 8.223
Appendix GGGGGGGGGG. Differences from Berkeley Pascal 8.224
Appendix HHHHHHHHHH. Differences from Berkeley Pascal 8.225
Appendix IIIIIIIIII. Differences from Berkeley Pascal 8.226
Appendix JJJJJJJJJJ. Differences from Berkeley Pascal 8.227
Appendix KKKKKKKKKK. Differences from Berkeley Pascal 8.228
Appendix LLLLLLLLLL. Differences from Berkeley Pascal 8.229
Appendix MMMMMMMMMM. Differences from Berkeley Pascal 8.230
Appendix NNNNNNNNNN. Differences from Berkeley Pascal 8.231
Appendix OOOOOOOOOO. Differences from Berkeley Pascal 8.232
Appendix PPPPPPPPPP. Differences from Berkeley Pascal 8.233
Appendix QQQQQQQQQQ. Differences from Berkeley Pascal 8.234
Appendix RRRRRRRRRR. Differences from Berkeley Pascal 8.235
Appendix SSSSSSSSSS. Differences from Berkeley Pascal 8.236
Appendix TTTTTTTTTT. Differences from Berkeley Pascal 8.237
Appendix UUUUUUUUUU. Differences from Berkeley Pascal 8.238
Appendix VVVVVVVVVV. Differences from Berkeley Pascal 8.239
Appendix WWWWWWWWWW. Differences from Berkeley Pascal 8.240
Appendix XXXXXXXXXX. Differences from Berkeley Pascal 8.241
Appendix YYYYYYYYYY. Differences from Berkeley Pascal 8.242
Appendix ZZZZZZZZZZ. Differences from Berkeley Pascal 8.243
Appendix AAAAAAAAAAA. Differences from Berkeley Pascal 8.244
Appendix BBBBBBBBBBB. Differences from Berkeley Pascal 8.245
Appendix CCCCCCCCCCC. Differences from Berkeley Pascal 8.246
Appendix DDDDDDDDDDD. Differences from Berkeley Pascal 8.247
Appendix EEEEEEEEEEE. Differences from Berkeley Pascal 8.248
Appendix FFFFFFFFFFF. Differences from Berkeley Pascal 8.249
Appendix GGGGGGGGGGG. Differences from Berkeley Pascal 8.250
Appendix HHHHHHHHHHH. Differences from Berkeley Pascal 8.251
Appendix IIIIIIIIIII. Differences from Berkeley Pascal 8.252
Appendix JJJJJJJJJJJ. Differences from Berkeley Pascal 8.253
Appendix KKKKKKKKKKK. Differences from Berkeley Pascal 8.254
Appendix LLLLLLLLLLL. Differences from Berkeley Pascal 8.255
Appendix MMMMMMMMMMM. Differences from Berkeley Pascal 8.256
Appendix NNNNNNNNNNN. Differences from Berkeley Pascal 8.257
Appendix OOOOOOOOOOO. Differences from Berkeley Pascal 8.258
Appendix PPPPPPPPPPP. Differences from Berkeley Pascal 8.259
Appendix QQQQQQQQQQQ. Differences from Berkeley Pascal 8.260
Appendix RRRRRRRRRRR. Differences from Berkeley Pascal 8.261
Appendix SSSSSSSSSSS. Differences from Berkeley Pascal 8.262
Appendix TTTTTTTTTTT. Differences from Berkeley Pascal 8.263
Appendix UUUUUUUUUUU. Differences from Berkeley Pascal 8.264
Appendix VVVVVVVVVVV. Differences from Berkeley Pascal 8.265
Appendix WWWWWWWWWWW. Differences from Berkeley Pascal 8.266
Appendix XXXXXXXXXXX. Differences from Berkeley Pascal 8.267
Appendix YYYYYYYYYYY. Differences from Berkeley Pascal 8.268
Appendix ZZZZZZZZZZZ. Differences from Berkeley Pascal 8.269
Appendix AAAAAAAAAAAA. Differences from Berkeley Pascal 8.270
Appendix BBBBBBBBBBBB. Differences from Berkeley Pascal 8.271
Appendix CCCCCCCCCCCC. Differences from Berkeley Pascal 8.272
Appendix DDDDDDDDDDDD. Differences from Berkeley Pascal 8.273
Appendix EEEEEEEEEEEE. Differences from Berkeley Pascal 8.274
Appendix FFFFFFFFFFFF. Differences from Berkeley Pascal 8.275
Appendix GGGGGGGGGGGG. Differences from Berkeley Pascal 8.276
Appendix HHHHHHHHHHHH. Differences from Berkeley Pascal 8.277
Appendix IIIIIIIIIIII. Differences from Berkeley Pascal 8.278
Appendix JJJJJJJJJJJJ. Differences from Berkeley Pascal 8.279
Appendix KKKKKKKKKKKK. Differences from Berkeley Pascal 8.280
Appendix LLLLLLLLLLLL. Differences from Berkeley Pascal 8.281
Appendix MMMMMMMMMMMM. Differences from Berkeley Pascal 8.282
Appendix NNNNNNNNNNNN. Differences from Berkeley Pascal 8.283
Appendix OOOOOOOOOOOO. Differences from Berkeley Pascal 8.284
Appendix PPPPPPPPPPPP. Differences from Berkeley Pascal 8.285
Appendix QQQQQQQQQQQQ. Differences from Berkeley Pascal 8.286
Appendix RRRRRRRRRRRR. Differences from Berkeley Pascal 8.287
Appendix SSSSSSSSSSSS. Differences from Berkeley Pascal 8.288
Appendix TTTTTTTTTTTT. Differences from Berkeley Pascal 8.289
Appendix UUUUUUUUUUUU. Differences from Berkeley Pascal 8.290
Appendix VVVVVVVVVVVV. Differences from Berkeley Pascal 8.291
Appendix WWWWWWWWWWWW. Differences from Berkeley Pascal 8.292
Appendix XXXXXXXXXXXX. Differences from Berkeley Pascal 8.293
Appendix YYYYYYYYYYYY. Differences from Berkeley Pascal 8.294
Appendix ZZZZZZZZZZZZ. Differences from Berkeley Pascal 8.295
Appendix AAAAAAAAAAAAA. Differences from Berkeley Pascal 8.296
Appendix BBBBBBBBBBBBB. Differences from Berkeley Pascal 8.297
Appendix CCCCCCCCCCCCC. Differences from Berkeley Pascal 8.298
Appendix DDDDDDDDDDDDD. Differences from Berkeley Pascal 8.299
Appendix EEEEEEEEEEEEE. Differences from Berkeley Pascal 8.300
Appendix FFFFFFFFFFFFF. Differences from Berkeley Pascal 8.301
Appendix GGGGGGGGGGGGG. Differences from Berkeley Pascal 8.302
Appendix HHHHHHHHHHHHH. Differences from Berkeley Pascal 8.303
Appendix IIIIIIIIIIIII. Differences from Berkeley Pascal 8.304
Appendix JJJJJJJJJJJJJ. Differences from Berkeley Pascal 8.305
Appendix KKKKKKKKKKKKK. Differences from Berkeley Pascal 8.306
Appendix LLLLLLLLLLLLL. Differences from Berkeley Pascal 8.307
Appendix MMMMMMMMMMMMM. Differences from Berkeley Pascal 8.308
Appendix NNNNNNNNNNNNN. Differences from Berkeley Pascal 8.309
Appendix OOOOOOOOOOOOO. Differences from Berkeley Pascal 8.310
Appendix PPPPPPPPPPPPP. Differences from Berkeley Pascal 8.311
Appendix QQQQQQQQQQQQQ. Differences from Berkeley Pascal 8.312
Appendix RRRRRRRRRRRRR. Differences from Berkeley Pascal 8.313
Appendix SSSSSSSSSSSSS. Differences from Berkeley Pascal 8.314
Appendix TTTTTTTTTTTTT. Differences from Berkeley Pascal 8.315
Appendix UUUUUUUUUUUUU. Differences from Berkeley Pascal 8.316
Appendix VVVVVVVVVVVVV. Differences from Berkeley Pascal 8.317
Appendix WWWWWWWWWWWWW. Differences from Berkeley Pascal 8.318
Appendix XXXXXXXXXXXXX. Differences from Berkeley Pascal 8.319
Appendix YYYYYYYYYYYYY. Differences from Berkeley Pascal 8.320
Appendix ZZZZZZZZZZZZZ. Differences from Berkeley Pascal 8.321
Appendix AAAAAAAAAAAAAA. Differences from Berkeley Pascal 8.322
Appendix BBBBBBBBBBBBBB. Differences from Berkeley Pascal 8.323
Appendix CCCCCCCCCCCCCC. Differences from Berkeley Pascal 8.324
Appendix DDDDDDDDDDDDDD. Differences from Berkeley Pascal 8.325
Appendix EEEEEEEEEEEEEE. Differences from Berkeley Pascal 8.326
Appendix FFFFFFFFFFFFFF. Differences from Berkeley Pascal 8.327
Appendix GGGGGGGGGGGGGG. Differences from Berkeley Pascal 8.328
Appendix HHHHHHHHHHHHHH. Differences from Berkeley Pascal 8.329
Appendix IIIIIIIIIIIIII. Differences from Berkeley Pascal 8.330
Appendix JJJJJJJJJJJJJJ. Differences from Berkeley Pascal 8.331
Appendix KKKKKKKKKKKKKK. Differences from Berkeley Pascal 8.332
Appendix LLLLLLLLLLLLLL. Differences from Berkeley Pascal 8.333
Appendix MMMMMMMMMMMMMM. Differences from Berkeley Pascal 8.334
Appendix NNNNNNNNNNNNNN. Differences from Berkeley Pascal 8.335
Appendix OOOOOOOOOOOOOO. Differences from Berkeley Pascal 8.336
Appendix PPPPPPPPPPPPPP. Differences from Berkeley Pascal 8.337
Appendix QQQQQQQQQQQQQQ. Differences from Berkeley Pascal 8.338
Appendix RRRRRRRRRRRRRR. Differences from Berkeley Pascal 8.339
Appendix SSSSSSSSSSSSSS. Differences from Berkeley Pascal 8.340
Appendix TTTTTTTTTTTTTT. Differences from Berkeley Pascal 8.341
Appendix UUUUUUUUUUUUUU. Differences from Berkeley Pascal 8.342
Appendix VVVVVVVVVVVVVV. Differences from Berkeley Pascal 8.343
Appendix WWWWWWWWWWWWWW. Differences from Berkeley Pascal 8.344
Appendix XXXXXXXXXXXXXX. Differences from Berkeley Pascal 8.345
Appendix YYYYYYYYYYYYYY. Differences from Berkeley Pascal 8.346
Appendix ZZZZZZZZZZZZZZ. Differences from Berkeley Pascal 8.347
Appendix AAAAAAAAAAAAAAA. Differences from Berkeley Pascal 8.348
Appendix BBBBBBBBBBBBBBB. Differences from Berkeley Pascal 8.349
Appendix CCCCCCCCCCCCCCC. Differences from Berkeley Pascal 8.350
Appendix DDDDDDDDDDDDDDD. Differences from Berkeley Pascal 8.351
Appendix EEEEEEEEEEEEEEE. Differences from Berkeley Pascal 8.352
Appendix FFFFFFFFFFFFFFF. Differences from Berkeley Pascal 8.353
Appendix GGGGGGGGGGGGGGG. Differences from Berkeley Pascal 8.354
Appendix HHHHHHHHHHHHHHH. Differences from Berkeley Pascal 8.355
Appendix IIIIIIIIIIIIIII. Differences from Berkeley Pascal 8.356
Appendix JJJJJJJJJJJJJJJ. Differences from Berkeley Pascal 8.357
Appendix KKKKKKKKKKKKKKK. Differences from Berkeley Pascal 8.358
Appendix LLLLLLLLLLLLLLL. Differences from Berkeley Pascal 8.359
Appendix MMMMMMMMMMMMMMM. Differences from Berkeley Pascal 8.360
Appendix NNNNNNNNNNNNNNN. Differences from Berkeley Pascal 8.361
Appendix OOOOOOOOOOOOOOO. Differences from Berkeley Pascal 8.362
Appendix PPPPPPPPPPPPPPP. Differences from Berkeley Pascal 8.363
Appendix QQQQQQQQQQQQQQQ. Differences from Berkeley Pascal 8.364
Appendix RRRRRRRRRRRRRRR. Differences from Berkeley Pascal 8.365
Appendix SSSSSSSSSSSSSSS. Differences from Berkeley Pascal 8.366
Appendix TTTTTTTTTTTTTTT. Differences from Berkeley Pascal 8.367
Appendix UUUUUUUUUUUUUUU. Differences from Berkeley Pascal 8.368
Appendix VVVVVVVVVVVVVVV. Differences from Berkeley Pascal 8.369
Appendix WWWWWWWWWWWWWWW. Differences from Berkeley Pascal 8.370
Appendix XXXXXXXXXXXXXXX. Differences from Berkeley Pascal 8.371
Appendix YYYYYYYYYYYYYYY. Differences from Berkeley Pascal 8.372
Appendix ZZZZZZZZZZZZZZZ. Differences from Berkeley Pascal 8.373
Appendix AAAAAAAAAAAAAAAA. Differences from Berkeley Pascal 8.374
Appendix BBBBBBBBBBBBBBBB. Differences from Berkeley Pascal 8.375
Appendix CCCCCCCCCCCCCCCC. Differences from Berkeley Pascal 8.376
Appendix DDDDDDDDDDDDDDDD. Differences from Berkeley Pascal 8.377
Appendix EEEEEEEEEEEEEEEE. Differences from Berkeley Pascal 8.378
Appendix FFFFFFFFFFFFFFFF. Differences from Berkeley Pascal 8.379
Appendix GGGGGGGGGGGGGGGG. Differences from Berkeley Pascal 8.380
Appendix HHHHHHHHHHHHHHHH. Differences from Berkeley Pascal 8.381
Appendix IIIIIIIIIIIIIIII. Differences from Berkeley Pascal 8.382
Appendix JJJJJJJJJJJJJJJJ. Differences from Berkeley Pascal 8.383
Appendix KKKKKKKKKKKKKKKK. Differences from Berkeley Pascal 8.384
Appendix LLLLLLLLLLLLLLLL. Differences from Berkeley Pascal 8.385
Appendix MMMMMMMMMMMMMMMM. Differences from Berkeley Pascal 8.386
Appendix NNNNNNNNNNNNNNNN. Differences from Berkeley Pascal 8.387
Appendix OOOOOOOOOOOOOOOO. Differences from Berkeley Pascal 8.388
Appendix PPPPPPPPPPPPPPPP. Differences from Berkeley Pascal 8.389
Appendix QQQQQQQQQQQQQQQQ. Differences from Berkeley Pascal 8.390
Appendix RRRRRRRRRRRRRRRR. Differences from Berkeley Pascal 8.391
Appendix SSSSSSSSSSSSSSSS. Differences from Berkeley Pascal 8.392
Appendix TTTTTTTTTTTTTTTT. Differences from Berkeley Pascal 8.393
Appendix UUUUUUUUUUUUUUUU. Differences from Berkeley Pascal 8.394
Appendix VVVVVVVVVVVVVVVV. Differences from Berkeley Pascal 8.395
Appendix WWWWWWWWWWWWWWWW. Differences from Berkeley Pascal 8.396
Appendix XXXXXXXXXXXXXXXX. Differences from Berkeley Pascal 8.397
Appendix YYYYYYYYYYYYYYYY. Differences from Berkeley Pascal 8.398
Appendix ZZZZZZZZZZZZZZZZ. Differences from Berkeley Pascal 8.399
Appendix AAAAAAAAAAAAAAAAA. Differences from Berkeley Pascal 8.400
Appendix BBBBBBBBBBBBBBBBB. Differences from Berkeley Pascal 8.401
Appendix CCCCCCCCCCCCCCCCC. Differences from Berkeley Pascal 8.402
Appendix DDDDDDDDDDDDDDDDD. Differences from Berkeley Pascal 8.403
Appendix EEEEEEEEEEEEEEEEE. Differences from Berkeley Pascal 8.404
Appendix FFFFFFFFFFFFFFFFF. Differences from Berkeley Pascal 8.405
Appendix GGGGGGGGGGGGGGGGG. Differences from Berkeley Pascal 8.406
Appendix HHHHHHHHHHHHHHHHH. Differences from Berkeley Pascal 8.407
Appendix IIIIIIIIIIIIIIIII. Differences from Berkeley Pascal 8.408
Appendix JJJJJJJJJJJJJJJJJ. Differences from Berkeley Pascal 8.409
Appendix KKKKKKKKKKKKKKKKK. Differences from Berkeley Pascal 8.410
Appendix LLLLLLLLLLLLLLLLL. Differences from Berkeley Pascal 8.411
Appendix MMMMMMMMMMMMMMMMM. Differences from Berkeley Pascal 8.412
Appendix NNNNNNNNNNNNNNNNN. Differences from Berkeley Pascal 8.413
Appendix OOOOOOOOOOOOOOOOO. Differences from Berkeley Pascal 8.414
Appendix PPPPPPPPPPPPPPPPP. Differences from Berkeley Pascal 8.415
Appendix QQQQQQQQQQQQQQQQQ. Differences from Berkeley Pascal 8.416
Appendix RRRRRRRRRRRRRRRRR. Differences from Berkeley Pascal 8.417
Appendix SSSSSSSSSSSSSSSSS. Differences from Berkeley Pascal 8.418
Appendix TTTTTTTTTTTTTTTTT. Differences from Berkeley Pascal 8.419
Appendix UUUUUUUUUUUUUUUUU. Differences from Berkeley Pascal 8.420
Appendix VVVVVVVVVVVVVVVVV. Differences from Berkeley Pascal 8.421
Appendix WWWWWWWWWWWWWWWWW. Differences from Berkeley Pascal 8.422
Appendix XXXXXXXXXXXXXXXXX. Differences from Berkeley Pascal 8.423
Appendix YYYYYYYYYYYYYYYYY. Differences from Berkeley Pascal 8.424
Appendix ZZZZZZZZZZZZZZZZZ. Differences from Berkeley Pascal 8.425
Appendix AAAAAAAAAAAAAAAAAA. Differences from Berkeley Pascal 8.426
Appendix BBBBBBBBBBBBBBBBBB. Differences from Berkeley Pascal 8.427
Appendix CCCCCCCCCCCCCCCCCC. Differences from Berkeley Pascal 8.428
Appendix DDDDDDDDDDDDDDDDDD. Differences from Berkeley Pascal 8.429
Appendix EEEEEEEEEEEEEEEEEE. Differences from Berkeley Pascal 8.430
Appendix FFFFFFFFFFFFFFFFFF. Differences from Berkeley Pascal 8.431
Appendix GGGGGGGGGGGGGGGGGG. Differences from Berkeley Pascal 8.432
Appendix HHHHHHHHHHHHHHHHHH. Differences from Berkeley Pascal 8.433
Appendix IIIIIIIIIIIIIIIIII. Differences from Berkeley Pascal 8.434
Appendix JJJJJJJJJJJJJJJJJJ. Differences from Berkeley Pascal 8.435
Appendix KKKKKKKKKKKKKKKKKK. Differences from Berkeley Pascal 8.436
Appendix LLLLLLLLLLLLLLLLLL. Differences from Berkeley Pascal 8.437
Appendix MMMMMMMMMMMMMMMMMM. Differences from Berkeley Pascal 8.438
Appendix NNNNNNNNNNNNNNNNNN. Differences from Berkeley Pascal 8.439
Appendix OOOOOOOOOOOOOOOOOO. Differences from Berkeley Pascal 8.440
Appendix PPPPPPPPPPPPPPPPPP. Differences from Berkeley Pascal 8.441
Appendix QQQQQQQQQQQQQQQQQQ. Differences from Berkeley Pascal 8.442
Appendix RRRRRRRRRRRRRRRRRR. Differences from Berkeley Pascal 8.443
Appendix SSSSSSSSSSSSSSSSSS. Differences from Berkeley Pascal 8.444
Appendix TTTTTTTTTTTTTTTTTT. Differences from Berkeley Pascal 8.445
Appendix UUUUUUUUUUUUUUUUUU. Differences from Berkeley Pascal 8.446
Appendix VVVVVVVVVVVVVVVVVV. Differences from Berkeley Pascal 8.447
Appendix WWWWWWWWWWWWWWWWWW. Differences from Berkeley Pascal 8.448
Appendix XXXXXXXXXXXXXXXXXX. Differences from Berkeley Pascal 8.449
Appendix YYYYYYYYYYYYYYYYYY. Differences from Berkeley Pascal 8.450
Appendix ZZZZZZZZZZZZZZZZZZ. Differences from Berkeley Pascal 8.451
Appendix AAAAAAAAAAAAAAAAAAA. Differences from Berkeley Pascal 8.452
Appendix BBBBBBBBBBBBBBBBBBB. Differences from Berkeley Pascal 8.453
Appendix CCCCCCCCCCCCCCCCCCC. Differences from Berkeley Pascal 8.454
Appendix DDDDDDDDDDDDDDDDDDD. Differences from Berkeley Pascal 8.455
Appendix EEEEEEEEEEEEEEEEEEE. Differences from Berkeley

We are now ready to execute the second example. The following output was produced by our first run.

[illegible]

Here the interpreter is presenting us with a runtime error diagnostic. It detected a 'division by zero' at line 17. Examining line 17, we see that we have written the statement 'x := d / i' instead of 'x := d * i'. We can correct this and rerun the program:

[illegible]

This appears to be the output we wanted. We could now save the output in a file if we wished by using the shell to redirect the output:

```
% px > graph
```

We can use *cat* (1) to see the contents of the file graph. We can also make a listing of the graph on the line printer without putting it into a file, e.g.

[illegible]

Neither that the statistics lines came out on our terminal. The statistics line comes out on the diagnostic output (unit 2.) There are two ways to get rid of the statistics line. We can redirect the statistics message to the printer using the syntax '|&' to the shell

```
% px |& lpr
```

or we can translate the program with the **p** option disabled on the command line as we did above. This will disable all post-mortem dumping including the statistics line, thus:

```
% px -p trigger.p
```

[illegible]

Discussion
time of last modification of the source file. Thus the *px* version of the program will be recompiled if the Pascal source file has been modified since the last time a version was prepared. The time given with the file name (also on the header line) indicates the

[illegible]

line 10 of the file. Note that the length of the program line is reported correctly, but that the displayed text is the original and not the corresponding text in the file. The vertical bar is labeled with a comment to indicate its significance, namely that lines past the column were truncated.
3. Error diagnostics information on *px* can be found in its manual section *px* (1) and in sections 5.4, 5.5 and 5.10.
Illegal characters The details of the diagnostics produced by *Pascal* will discuss here only the diagnostics of the programs *pi*, *pc* and *px*. *Pix* is a simple but useful program which invokes *pi* and *px* to do all the real processing. See its manual section *pix* (1) and section 5.2 below for

[illegible]

placed within curly braces. By using one kind of comment exclusively in your program you can use the other delimiters when you need to "comment out" parts of your program†. In this way you will also allow the translator to help by detecting statements accidentally

[illegible]

This part of the language is a minor nuisance. Pascal requires digits in real numbers both before and after the decimal point. Thus the following statements, which look quite reasonable to FORTRAN users, generate diagnostics in Pascal:

[illegible]

allow for simple alterations without the need to edit the input file. For example, one might want to alter the program text in response to different input data. Also, part of the input file may be changed while debugging a program. Consider the following example.

[illegible]

diagnostic which indicates that the translator is confused. In this case, recovery will examine the input to the left of the point of error to see if there is only one symbol which can follow this input. If this is the case, the recovery will print a

[illegible]

Consider the following example:

[illegible]

The message indicates that the translator is confused by the input text that it has received. In this case, it is not obvious that the input is an integer. The message will indicate that an end-of-file was expected. This situation may occur after a bracketing error, or if too many **end**s are present in the input.

[illegible]

3.2. Translator semantic errors

[illegible]

These words are plugged into a great number of error messages. Thus, if you tried to assign an *integer* value to a *char* variable you would receive a diagnostic like the following:

[illegible]

Scalar case, one error produced a two line error message. If the same error occurs more than once, the same explanatory diagnostic will be given each time.

[illegible]

```
type color = (red, green, blue)
```

Function and procedure parameters the notion *integer* is used. Although it would be correct, in the context of the *User Manual* to refer to an integer variable as a *scalar* variable *pi* prefers the more specific identification.
For built-in procedures and functions, two kinds of errors occur. If the routines are called with the wrong number of arguments a message similar to:

[illegible]

is given. If the type of the argument is wrong, a message like

[illegible]

Can't read and write scalars procedures implemented in Pascal 6000-3.4 are diagnosed as unimplemented in Berkeley Pascal, notably those related to **segmented** files.
The messages which state that scalar (user-defined) types cannot be written to and from files are often mysterious. It is in fact the case that if you define

```
type color = (red, green, blue)
```

[illegible]

```
file of color
```

Expression diagnostics binary rather than string.
The diagnostics for semantically ill-formed expressions are very explicit. Consider this sample translation:

[illegible]

Type equivalence identifier. Thus the declarations provided by the Pascal translator complain about 'non-equivalent types'. In general, Berkeley Pascal considers variables to have the same type only if they were declared with the same constructed type or with the same type

```
var y: ↑ integer;
```

do not have the same type. The assignment

```
x := y
```

thus produces the diagnostics:

[illegible]

Thus it is always necessary to declare a type such as

```
type intptr = ↑ integer;
```

and use it to declare

```
var x: intptr; y: intptr;
```

Note that if we had initially declared

```
var x, y: ↑ integer;
```

then the assignment statement would have worked. The statement

```
x↑ := y↑
```

Unreachable statements Since the parameter to a **procedure** or **function** must be declared with a type identifier rather than a constructed type, it is always necessary, in practice, to declare any type which will be used in this way.
diagnostic is produced for the unreachable statements. Such statements usually correspond to errors in the program logic. Note that a statement is considered to be reachable if there is a potential path of control, even if it can never be taken. Thus, no

```
if false then writeln('impossible!')
```

Goto's into structured statements
Unused and uninitialized variables [illegible] **goto** statements which transfer control into structured statements (**for**, **while**, etc.) It does not allow such jumps, nor does it allow branching from the **then** part of an **if** statement into the **else** part. Such help detect errors in the program logic, especially when a variable is local to a **procedure** and **function** entry. *pc* does not unless runtime checking is enabled using the **C** option. It is **not** good programming practice to rely on this initialization. To discourage this practice, and to
1) Use of a variable which is never assigned a value.
2) A variable which is declared but never used, distinguishing between those variables for which values are computed but which are never used, and those completely unused.
Panics in the translator [illegible] **panics**, are reported to all declared items. Thus a **const** or a **procedure** which is declared but never used is flagged. The **w** option of *pi* may be used to suppress these warnings; see sections 5.1 and 5.2.
One class of error which rarely occurs, but which causes termination of all processing when it does is a panic. A panic indicates a translator-detected internal inconsistency. A typical panic message is:

[illegible]

Out of memory [illegible] **out of space** message from the translator during translation of a large **procedure** or **function** or one containing a large number of string constants you may yet be able to translate your program if you break this one **procedure** or
3.4. Run-time errors Other errors which you may encounter when running *pi* relate to input-output. If *pi* cannot open the file you specify, or if the file is empty, you will be so informed.
Start-up errors We saw, in our second example, a run-time error. We here give the general description of run-time errors. The more unusual interpreter error messages are explained briefly in the manual section for *px* (1).
Program execution errors [illegible] occur when the object file to be executed is not available or appropriate. Typical errors here are caused by the specified object file not existing, not being a Pascal object, or being inaccessible to the user.
[illegible] As an example of such an error, assume that we have accidentally declared the constant *n1* to be 6, instead of 7 on line 2 of the program primes as given in section 2.6 above. If we run this program we get the following response.

[illegible]

Interrupts Here the interpreter indicates that the program terminated abnormally due to a subscript out of range near line 14, which is eight lines into the body of the program primes.
I/O interaction errors [illegible] program is interrupted while executing and the **p** option was not specified, then a backtrace will be printed.† The file *pmon.out* of profile information will be written if the program was translated with the **z** option enabled to *pi* or *pix*.
4. Input/output The final class of interpreter errors results from inappropriate interactions with files, including the user's terminal. Included here are bad formats for integer and real numbers (such as no digits after the decimal point) when reading.
4.1. Introduction [illegible] This section describes features of the Pascal input/output environment, with special consideration of the features peculiar to an interactive implementation.
[illegible] ing Pascal program which simply prints the message 'hi' [illegible] the file *output*. We gave examples there of redirecting the output to a file and to the line printer using the shell. Similarly, we can read the input from a file or another program. Consider the follow-

---

†The counts are completely accurate only in the absence of runtime errors and nonlocal **goto** statements. This is not generally a problem, however, as in structured programs nonlocal **goto** statements occur infrequently, and counts are incorrect after abnormal termination only when the *upward look* described below to get a count passes a suspended call point.

‡*Pmon.out* has a name similar to *mon.out* the monitor file produced by the profiling facility of the C compiler *cc* (1). See *prof* (1) for a discussion of the C compiler profiling facilities.

†If you wish to transport your program, especially to the 6000-3.4 implementation, you should use the character sequence '(*' to delimit comments. For transportation over the *rcslink* to Pascal 6000-3.4, the character '#' should be used to delimit characters and constant strings.

‡The checks for running out of memory are not foolproof and there is a chance that the interpreter will fault, producing a core image when it runs out of memory. This situation occurs very rarely.

∗ On the VAX-11, each variable is restricted to allocate at most 65000 bytes of storage (this is a PDP-11ism that has survived to the VAX.)

†Occasionally, the Pascal system will be in an inconsistent state when this occurs, e.g. when an interrupt terminates a **procedure** or **function** entry or exit. In this case, the backtrace will only contain the current line. A reverse call order list of procedures will not be given.

file now contains the output of the prime number program of section 2.6. It is also possible to 'pipe' input to this program much as we piped input to the line printer

```
% cat primes | pix –l kat.p
```

created the file *kat.out*. We have also used the shell to control the input to a program from files. One very simple way to associate Pascal files with named UNIX files is to place the file name in the **program** statement. For example, suppose we have previously

```
[illegible]
```

[illegible] time.

[illegible] file *dog.gin* in the current [illegible] UNIX. [illegible] the Pascal system to the [illegible] [illegible] [illegible]

```
% pi –pbt pxref.p
```

5.4. Options available in Px

and *argv* as described in section 4.6. *px* is the name of the file containing the program to be interpreted. If no arguments are given, then the file *obj* is executed. If more arguments are given, they are available to the Pascal program by using the built-ins *argc* and *argv* as described in section 4.6. *px* is the name of the file containing the program to be interpreted. If no arguments are given, then the file *obj* is executed. If more arguments are given, they are available to the Pascal program by using the built-ins *argc*
Px may also be invoked automatically. In this case, whenever a Pascal object file name is given as a command, the command will be executed with *px* prepended to it; that is

```
% obj primes
```

will be converted to read

```
% px obj primes
```

5.5. Options available in pc

Generate assembly language – S
The program is compiled and the assembly language output is left in file appended .s. Thus

```
% pc –S foo.p
```

Symbolic Debugger Information – g
Redirect the output file – o
The *name* argument after the **-o** is used as the name of the output file instead of *a.out*. Its typical use is to name the compiled program using the root of the file name. Thus:

```
% pc –o myprog myprog.p
```

Generate counters for gprof execution profile – p
Run the object file optimizer – O
5.6. Options available in Px
none of these options is specified, then *pxp* lists a list of procedures and functions in the program file name, which must end in '.p' as it must for *pi*, *pc*, and *pix*. *Pxp* will produce an execution profile if any of the **z**, **t** or **c** options is specified on the command line. If
It is important to note that only the **z** and **w** options of *pxp*, which are common to *pi*, *pc*, and *pxp* can be controlled in comments. All other options must be specified on the command line to have any effect.
Include the bodies of all routines in the profile – a
Suppress declaration parts from a profile – d
Eliminate include files in a profile – i
Left justify all procedure and function names – l
Enable and control the profiler – z

```
% pxp –z test parser.i compiler.p
```

5.7. Pxp
The program *pxp* can be used to reformat programs, by using a command of the form

```
% pxp dirty.p > clean.p
```

Note that since the shell creates the output file 'clean.p' before *pxp* executes, so 'clean.p' and 'dirty.p' must not be the same file.
preserving their
1) Left marginal comments, which begin in the first column of the input line and are placed in the first column of an output line.
2) Aligned comments, which are preceded by no input tokens on the input line. These are aligned in the output with the running program text.
3) Trailing comments, which are preceded in the input line by a token with no more than two spaces separating the token from the comment.
4) Right marginal comments, which are preceded in the input line by a token from which they are separated by at least three spaces or a tab. These are aligned down the right margin of the output, currently to the first tab stop after
Consider the following program.

```
{This is a left marginal comment, but past the margin}
```

When formatted by *pxp* the following output is produced.

```
{This is a left marginal comment}
```

Strip comments – s
Underline keywords – u
A command line argument of the form **–**_, as in

```
% pxp –_ dirty.p
```

Specify indenting unit – [2345678]
5.8. Pxref
The cross-reference program *pxref* may be used to make cross-referenced listings of Pascal programs. To produce a cross-reference of the program in the file 'foo.p' one can execute the command:

```
% pxref foo.p
```

5.9. Multiple file programs
in editing, listing, and maintenance. This facility allows the interpolation of source text from other files into the source stream of the translator. It can be used to divide large programs into more manageable pieces for ease
angle '<' or double angle '<<' brackets around the name, as in

```
{$include <globals.h>}
```

mortem backtrace. At this point the **include** file statement is forgotten. Note that the input file is positioned to the line following the **include** file. See the descriptions of the **i** option of *pi* in section 5.2 above; this can be used to control listing when **include** files are present.
When a non-trivial line is encountered in the source text after an **include** finishes, the 'popped' filename is printed, in the same manner as above.

5.10. Separately compiled programs
especially useful for large programs on local small machines with limited memory. This facility allows a number of files and the pieces to be compiled individually, to be linked together at some later time. This is
An example of the use of included **.X** files in a program would be:

```
{$include <globals.h>}
program main(input, output, obj);
```

scanner, parser. This might include the main program, the definition and declaration of all the global **label**s, **const**s, **type**s and **var**s from the file globals.h, and the **external function** and **procedure** declarations for each of the separately compiled files for the

```
function scanner(var inputfile: text): token;
```

Then the scanner might be in a separately compiled file containing:

```
{$include <globals.h>}
```

Standard Pascal
Modula/2
Berkeley Pascal
Differences between Berkeley Pascal and Standard Pascal
String padding
Berkeley Pascal will pad constant strings with blanks in expressions and as value parameters to make them as long as is required. The following is a legal Berkeley Pascal program:

```
program padded(output);
var z : array [ 1 .. 13 ] of char;
```

The padded blanks are added on the right. Thus the assignment above is equivalent to:

```
z := 'red  ';
```

Octal constants, octal and hexadecimal write
Octal constants may be given as a sequence of octal digits followed by the character 'b' or 'B'. The forms

```
write(a:n oct)
```

and

```
write(a:n hex)
```

Assert statement
abled. The syntax
assert <expr>

Enumerated type input-output
Structure initializations
Separate compilation
Extensions to Standard Pascal
Differences between Berkeley Pascal and Berkeley Pascal
Implementation limits
The program statement
Each Pascal file variable is associated with a named UNIX file. Except for *input* and *output*, which are exceptions to some of the rules, a name can become associated with a file in any of three ways:
1) If a global Pascal file variable appears in the **program** statement then it is associated with UNIX file of the same name.
2) If a file was reset or rewritten using the extended two-argument form of *reset* or *rewrite* then the given name is associated.
The syntax of the **program** statement is:

```
program <id> ( <file id> [ , <file id > ] ) ;
```

The files input and output
The formal parameters *input* and *output* are associated with the UNIX standard input and output and have a somewhat special status. The following rules must be noted:
1) The program heading **must** contains the formal parameter *output*. If *input* is used, explicitly or implicitly, then it must also be declared here.
2) Unlike all other files, the Pascal files *input* and *output* must not be defined in a declaration, as their declaration is automatically:

```
var input, output: text
```

3)

```
rewrite(output)
```

Details for files
Buffering
Binary files
Temporary files
Text files
Interactive I/O
Command line arguments
Options control
Options control
**Options control

as the first line of the program. The 'l' option is more appropriately specified on the command line, since it is extremely unlikely in an interactive environment that one wants a listing of the program each time it is translated.

Notes on the listing The first page of a listing includes a banner line indicating the version and date of generation of *pi* or *pc*. It also includes the UNIX path name supplied for the source file and the date of last modification of that file.

Restrictions and limitations Files may be declared only at the outermost level of variable declaration. Files may be passed as parameters only to procedures and functions which are declared **external**. Non-printing characters may not be written to text files. Text files may be read only character by character. Reading of numbers directly from text files is not allowed. Packed files of characters are not supported. Files of files are not allowed. Files may not be components of records or members of dynamically allocated structures.

Arrays, sets and strings Arrays may have at most 65535 elements. Each dimension of a multi-dimensional array is limited to 65535 elements. Sets may be defined. The lower bound of such a range must be greater than or equal to –32768, and the upper bound must be less than 32768. Packed arrays of characters are treated specially. Unpacked arrays of characters are not. Strings may be assigned to packed arrays of characters of the same length. Strings may be compared using the relational operators, with the shorter string padded on the right with blanks if necessary. Strings may be passed as value parameters to procedures and functions only if the formal parameter is packed array [1..n] of char, where n is the length of the actual string parameter. Strings may be passed as variable parameters only to formal parameters of exactly the same length. Non-printing characters may be written to text files, but they will be displayed in an implementation-dependent manner. Strings may not contain the character with ordinal value 0, which is used internally to mark the end of a string.

Procedures and functions as parameters Procedures and functions may be passed as parameters. Standard procedures and functions may not be passed as parameters. Nested procedures and functions may not be passed as parameters.

Identifiers Identifiers may be of any length. Only the first 8 characters are significant. Thus, for example, the identifiers 'abcdefgh' and 'abcdefghi' would be considered identical. Identifiers are considered to be distinct if they differ in any single position over their entire length. There is a limit, however, on the maximum input line length. This limit is quite generous and should rarely be encountered. Reserved words may not be redefined.

Labels Labels must be integers in the range 0 to 9999. Labels may be declared and referenced only in the procedure or function in which they appear.

Overflow On the VAX-11, there is an implementation defined limit of 65536 bytes per variable. There is no limit on the number of variables.

Additional predefined types The type *alfa* is predefined as:

```
type alfa = packed array [ 1..10 ] of char
```

The type *intset* is predefined as:

```
type intset = set of 0..127
```

Note that if *intset* is redefined via:

```
type intset = set of 0..58;
```

Additional predefined operations The relationals '<' and '>' of proper set inclusion are available. With *a* and *b* sets, note that

```
(not (a < b)) <> (a >= b)
```

does not hold for sets *a* = [0,2] and *b* = [1]. The only relation true between these sets is '<>'.

Nonstandard procedures

argv(i,a)	where *i* is an integer and *a* is a string variable assigns the (possibly truncated or blank padded) *i* 'th argument of the invocation of the current UNIX process to the variable *a*. The range of valid *i* is *0* to *argc–1*.
date(a)	assigns the current date to the alfa variable *a* in the format 'dd mmm yy ', where 'mmm' is the first three characters of the month, i.e. 'Apr'.
flush(f)	writes the output buffered for Pascal file *f* into the associated UNIX file.
halt	terminates the execution of the program with a control flow backtrace.
linelimit(f,x)‡	with *f* a textfile and *x* an integer expression causes the program to be abnormally terminated if more than *x* lines are written on file *f*. If *x* is less than 0 then no limit is imposed.
message(x,...)	causes the parameters, which have the format of those to the built-in **procedure** *write*, to be written unbuffered on the diagnostic unit 2, almost always the user's terminal.
null	a procedure of no arguments which does absolutely nothing. It is useful as a place holder, and is generated by *pxp* in place of the invisible empty statement.
remove(a)	where *a* is a string causes the UNIX file whose name is *a*, with trailing blanks eliminated, to be removed.
reset(f,a)	where *a* is a string causes the file whose name is *a* (with blanks trimmed) to be associated with *f* in addition to the normal function of *reset*.
rewrite(f,a)	is analogous to 'reset' above.
stlimit(i)	where *i* is an integer sets the statement limit to be *i* statements. Specifying the **p** option to *pc* disables statement limit counting.

Nonstandard functions

argc	returns the count of arguments when the Pascal program was invoked. *Argc* is always at least 1.
card(x)	returns the cardinality of the set *x*, i.e. the number of elements contained in the set.
clock	returns an integer which is the number of central processor milliseconds of user time used by this process.
expo(x)	yields the integer valued exponent of the floating-point representation of *x*; expo(*x*) = entier(log2(abs(*x*))).
random(x)	yields a real pseudo-random number in the range [0.0,1.0) and updates the seed value for the next call. The method used is a 46307 bit linear congruential generator. Successive seeds are generated as (seed*a + c) mod m and the new random number is a normalization of the seed value. The parameter *x* is ignored.
seed(i)	where *i* is an integer sets the random number generator seed to *i* and returns the previous seed. Thus seed(seed(i)) has no effect except to yield value *i*.
sysclock	an integer function of no arguments returns the number of central processor milliseconds of system time used by this process.
undefined(x)	a Boolean function. Its argument is a real number and it always returns false.

A few remarks on standard and nonstandard Pascal The function *time* of no arguments returns the time in seconds since 00:00:00 GMT January 1, 1970. Although this is convenient for many purposes, it is inappropriate for others. Programs which depend upon the time of day, the length of months, or other details of the calendar are notoriously non-portable. Worse, they are frequently wrong. Use of the nonstandard procedures and functions listed above will certainly make your program less portable, but they will make it more correct. With luck, you will be able to isolate all of the implementation dependencies in a few routines which will be easy to change when you transport your program to other Pascal installations. While certain system dependencies are bound to creep in, judicious design and programming practice can usually eliminate most of the problems.

Features not available in Berkeley Pascal Subtle differences may cause trouble when attempting to transport programs between this implementation and Pascal 6000–3.4. Using the **s** translator option may serve to indicate many problem areas.†

Features available in Berkeley Pascal but not in Pascal 6000-3.4

	Segmented files and associated functions and procedures.
	The function *trunc* with two arguments.
	The procedures *reset* and *rewrite* with file names.
	The functions *argc*, *seed*, *sysclock*, and *wallclock*.
	The procedures *argv*, *flush*, and *remove*.
	Message with arguments other than character strings.
	Write with keyword **hex**.
	The **assert** statement.
	Reading and writing of enumerated types.
	Allowing functions to return structures.
	Separate compilation of programs.

Other problem areas

	Comparison of records.
	Sets and strings are more general in Berkeley Pascal; see the restrictions given in the Jensen-Wirth *User Manual* for details on the 6000–3.4 restrictions.
	The character set differences may cause problems, especially the use of the function *chr*, characters as arguments to *ord*, and comparisons of characters, since the character set ordering differs between the two machines.

Berkeley Pascal 6000-3.4 compiler uses a less strict notion of type equivalence. In Berkeley Pascal, types are considered identical only if they are represented by the same type identifier. Thus, in particular, unnamed types are unique to the variables/fields declared. Pascal 6000–3.4 doesn't recognize our option flags, so it is wise to put the control of Berkeley Pascal options to the end of option lists or, better yet, restrict the option list length to one.

Implementation notes In Pascal 6000–3.4 the ordering of files in the program statement has significance. It is desirable to place *input* and *output* as the first two files in the **program** statement.

Acknowledgments The financial support of William Joy and Susan Graham by the National Science Foundation under grants MCS74-07644-A04, MCS78-07291, and MCS80-05144, and of William Joy by an IBM Graduate Fellowship are gratefully acknowledged.

†The character generated by a control-i indents to the next 'tab stop'. Tab stops are set every 8 columns in UNIX. Tabs thus provide a quick way of indenting in the program.

†The current translator makes a special case of the construct 'if ... in [...]' and enforces only the more lax restriction on 16 bit arithmetic given above in this case.

‡Currently ignored by pdp-11 *px*.

†The **s** option does not, however, check that identifiers differ in the first 8 characters. *Pi* and *pc* also do not check the semantics of **packed**.

A Portable Fortran 77 Compiler

S. I. Feldman

P. J. Weinberger

Bell Laboratories
Murray Hill, New Jersey 07974

J. Berkman

University of California
Berkeley, CA 94720

ABSTRACT

The Fortran language has been revised. The new language, known as Fortran 77, became an official American National Standard on April 3, 1978. We report here on a compiler and run-time system for the new extended language. It is believed to be the first complete Fortran 77 system to be implemented. This compiler is designed to be portable, to be correct and complete, and to generate code compatible with calling sequences produced by C compilers. In particular, this Fortran is quite usable on UNIX† systems. In this paper, we describe the language compiled, interfaces between procedures, and file formats assumed by the I/O system. Appendix A describes the Fortran 77 language extensions.

This is a standard Bell Laboratories document reproduced with minor modifications to the text. The Bell Laboratory's appendix on ''Differences Between Fortran 66 and Fortran 77'' has been changed to Appendix A, and a local appendix has been added. Appendix B contains a list of Fortran 77 references (some from the original Bell document and some added at Berkeley).

Revised September, 1985

† UNIX is a registered trademark of The Open Group in the U.S. and other countries.

1. INTRODUCTION

The Fortran language has been revised. The new language, known as Fortran 77, became an official American National Standard [1] on April 3, 1978. Fortran 77 supplants 1966 Standard Fortran [2]. We report here on a compiler and run-time system for the new extended language. The compiler and computation library were written by S.I.F., the I/O system by P.J.W. We believe ours to be the first complete Fortran 77 system to be implemented. This compiler is designed to be portable to a number of different machines, to be correct and complete, and to generate code compatible with calling sequences produced by compilers for the C language [3]. In particular, it is in use on UNIX systems. Two families of C compilers are in use at Bell Laboratories, those based on D. M. Ritchie's PDP-11 compiler [4] and those based on S. C. Johnson's portable C compiler [5]. This Fortran compiler can drive the second passes of either family. In this paper, we describe the language compiled, interfaces between procedures, and file formats assumed by the I/O system. We will describe implementation details in companion papers.

1.1. Usage

At present, versions of the compiler run on and compile for the PDP-11, the VAX-11/780, and the Interdata 8/32 UNIX systems. The command to run the compiler is

f77 *flags file . . .*

f77 is a general-purpose command for compiling and loading Fortran and Fortran-related files. EFL [6] and Ratfor [7] source files will be preprocessed before being presented to the Fortran compiler. C and assembler source files will be compiled by the appropriate programs. Object files will be loaded. (The **f77** and **cc** commands cause slightly different loading sequences to be generated, since Fortran programs need a few extra libraries and a different startup routine than do C programs.) The following file name suffixes are understood:

.f	Fortran source file
.F	Fortran source file
.e	EFL source file
.r	Ratfor source file
.c	C source file
.s	Assembler source file
.o	Object file

Arguments whose names end with **.f** are taken to be Fortran 77 source programs; they are compiled, and each object program is left on the file in the current directory whose name is that of the source with **.o** substituted for **.f**.

Arguments whose names end with **.F** are also taken to be Fortran 77 source programs; these are first processed by the C preprocessor before being compiled by **f77**.

Arguments whose names end with **.r** or **.e** are taken to be Ratfor or EFL source programs, respectively; these are first transformed by the appropriate preprocessor, then compiled by **f77**.

In the same way, arguments whose names end with **.c** or **.s** are taken to be C or assembly source programs and are compiled or assembled, producing a **.o** file.

The following flags are understood:

−c Compile but do not load. Output for **x.f**, **x.F**, **x.e**, **x.r**, **x.c**, or **x.s** is put on file **x.o**.

−d Used in debugging the compiler.

−g Have the compiler produce additional symbol table information for *dbx(1)*. This flag is incompatible with **−O**. See section 1.4 for more details.

−i2 On machines which support short integers, make the default integer constants and variables short (see section 2.14). (**−i4** is the standard value of this option). All logical quantities will

be short.

–m Apply the M4 macro preprocessor to each EFL or Ratfor source file before using the appropriate compiler.

–o *file* Put executable module on file *file* . (Default is **a.out**).

–onetrip or **–1**
Compile code that performs every **do** loop at least once (see section 2.12).

–p Generate code to produce usage profiles.

–pg Generate code in the manner of **–p**, but invoke a run-time recording mechanism that keeps more extensive statistics. See *gprof*(1).

–q Suppress printing of file names and program unit names during compilation.

–r8 Treat all floating point variables, constants, functions and intrinsics as double precision and all complex quantities as double complex. See section 2.17.

–u Make the default type of a variable **undefined** (see section 2.3).

–v Print the version number of the compiler and the name of each pass.

–w Suppress all warning messages.

–w66 Suppress warnings about Fortran 66 features used.

–C Compile code that checks that subscripts are within array bounds. For multi-dimensional arrays, only the equivalent linear subscript is checked.

–D*name*=*def*

–D*name* Define the *name* to the C preprocessor, as if by '#define'. If no definition is given, the name is defined as "1". (**.F** files only).

–E*str* Use the string *str* as an EFL option in processing **.e** files.

–F Ratfor, EFL, and **.F** source files are pre-processed into **.f** files, and those **.f** files are left on the disk without being compiled.

–I*dir* '#include' files whose names do not begin with '/' are always sought first in the directory of the *file* argument, then in directories named in **–I** options, then in directories on a standard list. (**.F** files only).

–N[**qxscn**]*nnn*
Make static tables in the compiler bigger. The compiler will complain if it overflows its tables and suggest you apply one or more of these flags. These flags have the following meanings:

- **q** Maximum number of equivalenced variables. Default is 150.
- **x** Maximum number of external names (common block names, subroutine and function names). Default is 200.
- **s** Maximum number of statement numbers. Default is 401.
- **c** Maximum depth of nesting for control statements (e.g. DO loops). Default is 20.
- **n** Maximum number of identifiers. Default is 1009.

–O Invoke the object code optimizer. Incompatible with **–g**.

–R*str* Use the string *str* as a Ratfor option in processing **.r** files.

–U Do not convert upper case letters to lower case. The default is to convert Fortran programs to lower case except within character string constants.

–S Generate assembler output for each source file, but do not assemble it. Assembler output for a source file **x.f**, **x.F**, **x.e**, **x.r**, or **x.c** is put on file **x.s**.

Other flags, all library names (arguments beginning **–l**), and any names not ending with one of the understood suffixes are passed to the loader.

1.2. Documentation Conventions

In running text, we write Fortran keywords and other literal strings in boldface lower case. Examples will be presented in lightface lower case. Names representing a class of values will be printed in italics.

1.3. Implementation Strategy

The compiler and library are written entirely in C. The compiler generates C compiler intermediate code. Since there are C compilers running on a variety of machines, relatively small changes will make this Fortran compiler generate code for any of them. Furthermore, this approach guarantees that the resulting programs are compatible with C usage. The runtime computational library is complete. The runtime I/O library makes use of D. M. Ritchie's Standard C I/O package [8] for transferring data. With the few exceptions described below, only documented calls are used, so it should be relatively easy to modify to run on other operating systems.

1.4. Debugging Aids

A memory image is sometimes written to a file **core** in the current directory upon abnormal termination for errors caught by the **f77** libraries, user calls to **abort**, and certain signals (see **sigvec**(2) in the *UNIX Programmer's Manual*). **Core** is normally created only if the **–g** flag was specified to **f77** during loading.† The source-level debugger *dbx*(1) may be used with the executable and the **core** file to examine the image and determine what went wrong.

In the event that it is necessary to override this default behavior, the user may set the environment variable **f77_dump_flag**. If **f77_dump_flag** is set to a value beginning with **n**, a **core** file is not produced regardless of whether **–g** was specified at compile time, and if the value begins with **y**, dumps are produced even if **–g** was not specified.

2. LANGUAGE EXTENSIONS

Fortran 77 includes almost all of Fortran 66 as a subset. We describe the differences briefly in Appendix A. The most important additions are a character string data type, file-oriented input/output statements, and random access I/O. Also, the language has been cleaned up considerably.

In addition to implementing the language specified in the new Standard, our compiler implements a few extensions described in this section. Most are useful additions to the language. The remainder are extensions to make it easier to communicate with C procedures or to permit compilation of old (1966 Standard) programs.

2.1. Double Complex Data Type

The new type **double complex** is defined. Each datum is represented by a pair of double precision real values. The statements

```
z1 = ( 0.1d0, 0.2d0 )
z2 = dcmplx( dx, dy )
```

assign double complex values to **z1** and **z2**. The double precision values which constitute the double complex value may be isolated by using **dreal** or **dble** for the real part and **imag** or **dimag** for the imaginary part. To compute the double complex conjugate of a double complex value, use **conjg** or **dconjg**. The other **double complex** intrinsic functions may be accessed using their generic names or specific names. The generic names are: **abs**, **sqrt**, **exp**, **log**, **sin**, and **cos**. The specific names are the same as the generic names preceded by either **cd** or **z**, e.g. you may code **sqrt**, **zsqrt** or **cdsqrt** to compute the square root of a double complex value.

2.2. Internal Files

The Fortran 77 standard introduces ''internal files'' (memory arrays), but restricts their use to formatted sequential I/O statements. Our I/O system also permits internal files to be used in formatted direct reads

†Specify **–g** when loading with **cc** or **f77**; specify **–lg** as a library when using **ld** directly.

and writes and list directed sequential read and writes.

2.3. Implicit Undefined Statement

Fortran 66 has a fixed rule that the type of a variable that does not appear in a type statement is **integer** if its first letter is **i, j, k, l, m** or **n**, and **real** otherwise. Fortran 77 has an **implicit** statement for overriding this rule. As an aid to good programming practice, we permit an additional type, **undefined.** The statement

```
implicit undefined(a-z)
```

turns off the automatic data typing mechanism, and the compiler will issue a diagnostic for each variable that is used but does not appear in a type statement. Specifying the **–u** compiler flag is equivalent to beginning each procedure with this statement.

2.4. Recursion

Procedures may call themselves, directly or through a chain of other procedures. Since Fortran variables are by default **static**, it is often necessary to use the **automatic** storage extension to prevent unexpected results from recursive functions.

2.5. Automatic Storage

Two new keywords are recognized, **static** and **automatic.** These keywords may appear as ‘‘types’’ in type statements and in **implicit** statements. Local variables are static by default; there is only one instance of the variable. For variables declared **automatic,** there is a separate instance of the variable for each invocation of the procedure. Automatic variables may not appear in **equivalence, data,** or **save** statements. Neither type of variable is guaranteed to retain its value between calls to a subprogram (see the **save** statement in Appendix A).

2.6. Source Input Format

The Standard expects input to the compiler to be in 72-column format: except in comment lines, the first five characters are the statement number, the next is the continuation character, and the next 66 are the body of the line. (If there are fewer than 72 characters on a line, the compiler pads it with blanks; characters after the seventy-second are ignored.)

In order to make it easier to type Fortran programs, our compiler also accepts input in variable length lines. An ampersand ‘‘&’’ in the first position of a line indicates a continuation line; the remaining characters form the body of the line. A tab character in one of the first six positions of a line signals the end of the statement number and continuation part of the line; the remaining characters form the body of the line. A tab elsewhere on the line is treated as another kind of blank by the compiler.

In the Standard, there are only 26 letters — Fortran is a one-case language. Consistent with ordinary UNIX system usage, our compiler expects lower case input. By default, the compiler converts all upper case characters to lower case except those inside character constants. However, if the **–U** compiler flag is specified, upper case letters are not transformed. In this mode, it is possible to specify external names with upper case letters in them, and to have distinct variables differing only in case. If **–U** is specified, keywords will only be recognized in lower case.

2.7. Include Statement

The statement

```
include 'stuff'
```

is replaced by the contents of the file **stuff**; **include** statements may be nested to a reasonable depth, currently ten.

2.8. Binary Initialization Constants

A variable may be initialized in a **data** statement by a binary constant, denoted by a letter followed by a quoted string. If the letter is **b**, the string is binary, and only zeroes and ones are permitted. If the letter is **o**, the string is octal, with digits **0–7**. If the letter is **z** or **x**, the string is hexadecimal, with digits **0–9**, **a–f**. Thus, the statements

```
integer a(3)
data a / b'1010', o'12', z'a' /
```

initialize all three elements of **a** to ten.

2.9. Character Strings

For compatibility with C usage, the following backslash escapes are recognized:

\n	newline
\t	tab
\b	backspace
\f	form feed
\0	null
\'	apostrophe (does not terminate a string)
\"	quotation mark (does not terminate a string)
\\	\
\x	*x*, where *x* is any other character

Fortran 77 only has one quoting character, the apostrophe. Our compiler and I/O system recognize both the apostrophe " ' " and the double-quote " " ". If a string begins with one variety of quote mark, the other may be embedded within it without using the repeated quote or backslash escapes.

Each character string constant appearing outside a **data** statement is followed by a null character to ease communication with C routines.

2.10. Hollerith

Fortran 77 does not have the old Hollerith "*n* **h**" notation, though the new Standard recommends implementing the old Hollerith feature in order to improve compatibility with old programs. In our compiler, Hollerith data may be used in place of character string constants, and may also be used to initialize non-character variables in **data** statements.

2.11. Equivalence Statements

As a very special and peculiar case, Fortran 66 permits an element of a multiply-dimensioned array to be represented by a singly-subscripted reference in **equivalence** statements. Fortran 77 does not permit this usage, since subscript lower bounds may now be different from 1. Our compiler permits single subscripts in **equivalence** statements, under the interpretation that all missing subscripts are equal to 1. A warning message is printed for each such incomplete subscript.

2.12. One-Trip DO Loops

The Fortran 77 Standard requires that the range of a **do** loop not be performed if the initial value is already past the limit value, as in

```
do 10 i = 2, 1
```

The 1966 Standard stated that the effect of such a statement was undefined, but it was common practice that the range of a **do** loop would be performed at least once. In order to accommodate old programs, though they were in violation of the 1966 Standard, the **–onetrip** or **–1** compiler flags causes non-standard loops to be generated.

2.13. Commas in Formatted Input

The I/O system attempts to be more lenient than the Standard when it seems worthwhile. When doing a formatted read of non-character variables, commas may be used as value separators in the input record, overriding the field lengths given in the format statement. Thus, the format

(i10, f20.10, i4)

will read the record

−345,.05e−3,12

correctly.

2.14. Short Integers

On machines that support halfword integers, the compiler accepts declarations of type **integer∗2.** (Ordinary integers follow the Fortran rules about occupying the same space as a real variable; they are assumed to be of C type **long int**; halfword integers are of C type **short int**.) An expression involving only objects of type **integer∗2** is of that type. Generic functions return short or long integers depending on the actual types of their arguments. If a procedure is compiled using the **−i2** flag, all small integer constants will be of type **integer∗2.** If the precision of an integer-valued intrinsic function is not determined by the generic function rules, one will be chosen that returns the prevailing length (**integer∗2** when the **−i2** command flag is in effect). When the **−i2** option is in effect, all quantities of type **logical** will be short. Note that these short integer and logical quantities do not obey the standard rules for storage association.

2.15. Additional Intrinsic Functions

This compiler supports all of the intrinsic functions specified in the Fortran 77 Standard. In addition, there are built-in functions for performing bitwise logical and boolean operations on integer and logical values (**or**, **and**, **xor**, **not**, **lshift**, and **rshift**), and intrinsic functions for **double complex** values (see section 2.1). The **f77** library contains many other functions, such as accessing the UNIX command arguments (**getarg** and **iargc**) and environment (**getenv**). See **intro**(3f) and **bit**(3f) in the *UNIX Programmer's Manual* for more information.

2.16. Namelist I/O

Namelist I/O provides an easy way to input and output information without formats. Although not part of the standard, namelist I/O was part of many Fortran 66 systems and is a common extension to Fortran 77 systems.

Variables and arrays to be used in namelist I/O are declared as part of a namelist in a **namelist** statement, e.g.:

```
character str∗12
logical flags(20)
complex c(2)
real arr1(2,3), arr2(0:3,4)
namelist /basic/  arr1, arr2, key, str, c /flglst/ key, flags
```

This defines two namelists: list **basic** consists of variables **key** and **str** and arrays **arr1**, **arr2**, and **c**; list **flglst** consists of variable **key** and array **flags**. A namelist can include variables and arrays of any type, and a variable or array may be in several different namelists. However dummy arguments and array elements may not be in a namelist. A namelist name may be used in external sequential **read**, **write** and **print** statements wherever a format could be used.

In a namelist **read**, column one of each data record is ignored. The data begins with an ampersand in column 2 followed by the namelist name and a blank. Then there is a sequence of value assignments separated by commas and finally an ''&end''. A simple example of input data corresponding to namelist **basic** is:

```
&basic key=5, str='hi there' &end
```

For compatibility with other systems, dollar signs may be used instead of the ampersands:

```
$basic key=5, str='hi there' $end
```

A value assignment in the data record must be one of three forms. The simplest is a variable name followed by an equal sign followed by a data value which is assigned to that variable, e.g. ''key=5''. The second form consists of an array name followed by ''='' followed by one or more values to be assigned to the array, e.g.:

```
c=(1.1,–2.9),(–1.8e+10,14.0e–3)
```

assigns values to c(1) and c(2) in the complex array c.

As in other **read** statements, values are assigned in the order of the array in memory, i.e. column-major order for two dimensional arrays. Multiple copies of a value may be represented by a repetition count followed by an asterisk followed by the value; e.g. ''3*55.4'' is the same as ''55.4, 55.4, 55.4''. It is an error to specify more values than the array can hold; if less are specified, only that number of elements of the array are changed. The third form of a value assignment is a subscripted variable name followed by ''='' followed by a value or values, e.g.: ''arr2(0,4)=15.2''. Only integer constant subscripts may be used. The correct number of subscripts must be used and the subscripts must be legal. This form is the same as the form with an array name except the array is filled starting at the named element.

In all three forms, the variable or array name must be declared in the namelist. The form of the data values is the same as in list directed input except that in namelist I/O, character strings in the data must be enclosed in apostrophes or double quotes, and repetition counts must be followed by data values.

One use of namelist input is to read in a list of options or flags. For example:

```
      logical flags(14)
      namelist /pars/ flags, iters, xlow, xhigh, xinc
      data flags/14*.false./

10    read(5,pars,end=900)
      print pars
      call calc( xlow, xhigh, xinc, flags, iters )
      go to 10
900   continue
      end
```

could be run with the following data (each record begins with a space):

```
&pars iters=10, xlow=0.0, xhigh=1.0, xinc=0.1 &end
&pars xinc=0.2,
  flags(2)=2*.true., flags(8)=.true. &end
&pars xlow=2.0, xhigh=8.0 &end
```

The program reads parameters for the run from the first data set and computes using them. Then it loops and each successive set of namelist input data specifies only those data items which need to be changed. Note the second data set sets the 2^{nd}, 3^{rd}, and 8^{th} elements in the array **flags** to **.true.**.

When a namelist name is used in a **write** or **print** statement, all the values in the namelist are output together with their names. For example the **print** in the program above prints the following:

```
&pars flags= f, f, f, f, f, f, f, f, f, f, f, f, f, f, iters=
  10, xlow=  0., xhigh=   1.00000, xinc=  0.100000
&end
&pars flags= f, t, t, f, f, f, f, t, f, f, f, f, f, f, iters=
  10, xlow=  0., xhigh=   1.00000, xinc=  0.200000
&end
&pars flags= f, t, t, f, f, f, f, t, f, f, f, f, f, f, iters=
  10, xlow=   2.00000, xhigh=   8.00000, xinc=  0.200000
&end
```

Each line begins with a space so that namelist output can be used as input to a namelist **read**. The default is to use ampersands in namelist **print** and **write**. However, dollar signs will be used if the last preceding namelist **read** data set used dollar signs. The character to be used is stored as the first character of the common block **namelistkey**.

2.17. Automatic Precision Increase

The **−r8** flag allows a user to run a program with increased precision without changing any of the program source, i.e. it allows a user to take a program coded in single precision and compile and execute it as if it had been coded in double precision. The option extends the precision of all single precision real and complex constants, variables, external functions, and intrinsic functions. For example, the source:

```
implicit complex(c)
real last
intrinsic sin, csin
data last/0.3/

x = 0.1
y = sqrt(x)+sqrt(last)
c1 = (0.1,0.2)
c2 = sqrt(c1)
x = real(i)
y = aimag(c1)
call fun(sin,csin)
```

is compiled under this flag as if it had been written as:

```
implicit double precision (a-b,d-h,o-z), double complex(c)
double precision last
intrinsic dsin, cdsin
data last/0.3d0/

x = 0.1d0
y = sqrt(x)+sqrt(last)
c1 = (0.1d0,0.2d0)
c2 = sqrt(c1)
x = dreal(i)
y = dimag(c1)
call fun(dsin,cdsin)
```

When the **−r8** flag is invoked, the calls using the generic name **sqrt** will refer to a different specific function since the types of the arguments have changed. This option extends the precision of all single precision **real** and **complex** variables and functions, including those declared **real∗4** and **complex∗8**.

In order to successfully use this flag to increase precision, the entire program including all the subroutines and functions it calls must be recompiled. Programs which use dynamic memory allocation or use equivalence or common statements to associate variables of different types may have to be changed by hand. Similar caveats apply to the sizes of records in unformatted I/O.

2.18. Characters and Integers

A character constant of integer length or less may be assigned to an integer variable. Individual bytes are packed into the integer in the native byte order. The character constant is padded with blanks to the width of the integer during the assignment. Use of this feature is deprecated; it is intended only as a porting aid for extended Fortran 66 programs. Note that the intrinsic **ichar** function behaves as the standard requires, converting only single bytes to integers.

3. VIOLATIONS OF THE STANDARD

We know only a few ways in which our Fortran system violates the new standard:

3.1. Double Precision Alignment

The Fortran Standards (both 1966 and 1977) permit **common** or **equivalence** statements to force a double precision quantity onto an odd word boundary, as in the following example:

```
real a(4)
double precision b,c

equivalence (a(1),b), (a(4),c)
```

Some machines (e.g., Honeywell 6000, IBM 360) require that double precision quantities be on double word boundaries; other machines (e.g., IBM 370), run inefficiently if this alignment rule is not observed. It is possible to tell which equivalenced and common variables suffer from a forced odd alignment, but every double precision argument would have to be assumed on a bad boundary. To load such a quantity on some machines, it would be necessary to use separate operations to move the upper and lower halves into the halves of an aligned temporary, then to load that double precision temporary; the reverse would be needed to store a result. We have chosen to require that all double precision real and complex quantities fall on even word boundaries on machines with corresponding hardware requirements, and to issue a diagnostic if the source code demands a violation of the rule.

3.2. Dummy Procedure Arguments

If any argument of a procedure is of type character, all dummy procedure arguments of that procedure must be declared in an **external** statement. This requirement arises as a subtle corollary of the way we represent character string arguments and of the one-pass nature of the compiler. A warning is printed if a dummy procedure is not declared **external.** Code is correct if there are no **character** arguments.

3.3. T and TL Formats

The implementation of the **t** (absolute tab) and **tl** (leftward tab) format codes is defective. These codes allow rereading or rewriting part of the record which has already been processed (section 6.3.2 in Appendix A). The implementation uses seeks, so if the unit is not one which allows seeks, such as a terminal, the program is in error. A benefit of the implementation chosen is that there is no upper limit on the length of a record, nor is it necessary to predeclare any record lengths except where specifically required by Fortran or the operating system.

3.4. Carriage Control

The Standard leaves as implementation dependent which logical unit(s) are treated as "printer" files. In this implementation there is no printer file and thus by default, no carriage control is recognized on formatted output. This can be changed using **form='print'** in the **open** statement for a unit, or by using the **fpr**(1) filter for output; see [9].

3.5. Assigned Goto

The optional *list* associated with an assigned **goto** statement is not checked against the actual assigned value during execution.

4. INTER-PROCEDURE INTERFACE

To be able to write C procedures that call or are called by Fortran procedures, it is necessary to know the conventions for procedure names, data representation, return values, and argument lists that the compiled code obeys.

4.1. Procedure Names

On UNIX systems, the name of a common block or a Fortran procedure has an underscore appended to it by the compiler to distinguish it from a C procedure or external variable with the same user-assigned name. Fortran built-in procedure names have embedded underscores to avoid clashes with user-assigned subroutine names.

4.2. Data Representations

The following is a table of corresponding Fortran and C declarations:

Fortran	**C**
integer∗2 x	short int x;
integer x	long int x;
logical x	long int x;
real x	float x;
double precision x	double x;
complex x	struct { float r, i; } x;
double complex x	struct { double dr, di; } x;
character∗6 x	char x[6];

(By the rules of Fortran, **integer, logical,** and **real** data occupy the same amount of memory.)

4.3. Arrays

The first element of a C array always has subscript zero, while Fortran arrays begin at 1 by default. Fortran arrays are stored in column-major order in contiguous storage, C arrays are stored in row-major order. Many mathematical libraries have subroutines which transpose a two dimensional matrix, e.g. **f01crf** in the **NAG** library and **vtran** in the **IMSL** library. These may be used to transpose a two-dimensional array stored in C in row-major order to Fortran column-major order or vice-versa.

4.4. Return Values

A function of type **integer, logical, real,** or **double precision** declared as a C function returns the corresponding type. A **complex** or **double complex** function is equivalent to a C routine with an additional initial argument that points to the place where the return value is to be stored. Thus,

```
complex function f( . . . )
```

is equivalent to

```
f_(temp, . . .)
struct { float r, i; } *temp;
 . . .
```

A character-valued function is equivalent to a C routine with two extra initial arguments: a data address and a length. Thus,

```
character*15 function g( . . . )
```

is equivalent to

```
g_(result, length, . . .)
char result[ ];
long int length;
 . . .
```

and could be invoked in C by

```
char chars[15];
 . . .
g_(chars, 15L, . . . );
```

Subroutines are invoked as if they were **integer**-valued functions whose value specifies which alternate return to use. Alternate return arguments (statement labels) are not passed to the function, but are used to do an indexed branch in the calling procedure. (If the subroutine has no entry points with alternate return arguments, the returned value is undefined.) The statement

```
call nret(*1, *2, *3)
```

is treated exactly as if it were the computed **goto**

```
goto (1, 2, 3),  nret( )
```

4.5. Argument Lists

All Fortran arguments are passed by address. In addition, for every argument that is of type character or that is a dummy procedure, an argument giving the length of the value is passed. (The string lengths are **long int** quantities passed by value.) The order of arguments is then:

Extra arguments for complex and character functions
Address for each datum or function
A **long int** for each character or procedure argument

Thus, the call in

```
external f
character*7 s
integer b(3)
 . . .
call sam(f, b(2), s)
```

is equivalent to that in

```
int f();
char s[7];
long int b[3];
 . . .
sam_(f, &b[1], s, 0L, 7L);
```

4.6. System Interface

To run a Fortran program, the system invokes a small C program which first initializes signal handling, then calls **f_init** to initialize the Fortran I/O library, then calls your Fortran main program, and then calls **f_exit** to close any Fortran files opened.

f_init initializes Fortran units 0, 5, and 6 to standard error, standard input, and standard output respectively. It also calls **setlinebuf** to initiate line buffering of standard error. If you are using Fortran subroutines which may do I/O and you have a C main program, call **f_init** before calling the Fortran subroutines. Otherwise, Fortran units 0, 5, and 6 will be connected to files **fort.0**, **fort.5**, and **fort.6**, and error messages from the **f77** libraries will be written to **fort.0** instead of to standard error. If your C program terminates by calling the C function **exit**, all files are automatically closed. If there are Fortran scratch files to be deleted, first call **f_exit**. **F_init** and **f_exit** do not have any arguments.

The **–d** flag will show what libraries are used in loading Fortran programs.

5. FILE FORMATS

5.1. Structure of Fortran Files

Fortran requires four kinds of external files: sequential formatted and unformatted, and direct formatted and unformatted. On UNIX systems, these are all implemented as ordinary files which are assumed to have the proper internal structure.

Fortran I/O is based on *records*. When a direct file is opened in a Fortran program, the record length of the records must be given, and this is used by the Fortran I/O system to make the file look as if it is made up of records of the given length. In the special case that the record length is given as 1, the files are not considered to be divided into records, but are treated as byte-addressable byte strings; that is, as ordinary UNIX file system files. (A read or write request on such a file keeps consuming bytes until satisfied, rather than being restricted to a single record.)

The peculiar requirements on sequential unformatted files make it unlikely that they will ever be read or written by any means except Fortran I/O statements. Each record is preceded and followed by an integer containing the record's length in bytes.

The Fortran I/O system breaks sequential formatted files into records while reading by using each newline as a record separator. The result of reading off the end of a record is undefined according to the Standard. The I/O system is permissive and treats the record as being extended by blanks. On output, the I/O system will write a newline at the end of each record. It is also possible for programs to write newlines for themselves. This is an error, but the only effect will be that the single record the user thought he wrote will be treated as more than one record when being read or backspaced over.

5.2. Portability Considerations

The Fortran I/O system uses only the facilities of the standard C I/O library, a widely available and fairly portable package, with the following two nonstandard features: the I/O system needs to know whether a file can be used for direct I/O, and whether or not it is possible to backspace. Both of these facilities are implemented using the **fseek** routine, so there is a routine **canseek** which determines if **fseek** will have the desired effect. Also, the **inquire** statement provides the user with the ability to find out if two files are the same, and to get the name of an already opened file in a form which would enable the program to reopen it. Therefore there are two routines which depend on facilities of the operating system to provide these two services. In any case, the I/O system runs on the PDP-11, VAX-11/780, and Interdata 8/32 UNIX systems.

5.3. Logical Units and Files

Fortran logical unit numbers may be any integer between 0 and 99. The number of simultaneously open files is currently limited to 48.

Units 5, 6, and 0 are connected before the program begins to standard input, standard output, and standard error respectively.

If an unit is opened explicitly by an **open** statement with a **file=** keyword, then the file name is the name from the **open** statement. Otherwise, the default file name corresponding to unit *n* is **fort.***n*. If there is an environment variable whose name is the same as the tail of the file name after periods are deleted, then the contents of that environment variable are used as the name of the file. See [9] for details.

The default connection for all units is for sequential formatted I/O. The Standard does not specify where a file which has been explicitly **open**ed for sequential I/O is initially positioned. The I/O system will position the file at the beginning. Therefore a **write** will destroy any data already in the file, but a **read** will work reasonably. To position a file to its end, use a **read** loop, or the system dependent function **fseek**. The preconnected units 0, 5, and 6 are positioned as they come from the program's parent process.

APPENDIX A: Differences Between Fortran 66 and Fortran 77

The following is a very brief description of the differences between the 1966 [2] and the 1977 [1] Standard languages. We assume that the reader is familiar with Fortran 66. We do not pretend to be complete, precise, or unbiased, but plan to describe what we feel are the most important aspects of the new language. The best current information on the 1977 Standard is in publications of the X3J3 Subcommittee of the American National Standards Institute, and the ANSI X3.9-1978 document, the official description of the language. The Standard is written in English rather than a meta-language, but it is forbidding and legalistic. A number of tutorials and textbooks are available (see Appendix B).

1. Features Deleted from Fortran 66

1.1. Hollerith

All notions of "Hollerith" (*n* **h**) as data have been officially removed, although our compiler, like almost all in the foreseeable future, will continue to support this archaism.

1.2. Extended Range of DO

In Fortran 66, under a set of very restrictive and rarely-understood conditions, it is permissible to jump out of the range of a **do** loop, then jump back into it. Extended range has been removed in the Fortran 77 language. The restrictions are so special, and the implementation of extended range is so unreliable in many compilers, that this change really counts as no loss.

2. Program Form

2.1. Blank Lines

Completely blank lines are now legal comment lines.

2.2. Program and Block Data Statements

A main program may now begin with a statement that gives that program an external name:

```
program work
```

Block data procedures may also have names.

```
block data stuff
```

There is now a rule that only *one* unnamed block data procedure may appear in a program. (This rule is not enforced by our system.) The Standard does not specify the effect of the program and block data names, but they are clearly intended to aid conventional loaders.

2.3. ENTRY Statement

Multiple entry points are now legal. Subroutine and function subprograms may have additional entry points, declared by an **entry** statement with an optional argument list.

```
entry extra(a, b, c)
```

Execution begins at the first statement following the **entry** line. All variable declarations must precede all executable statements in the procedure. If the procedure begins with a **subroutine** statement, all entry points are subroutine names. If it begins with a **function** statement, each entry is a function entry point, with type determined by the type declared for the entry name. If any entry is a character-valued function, then all entries must be. In a function, an entry name of the same type as that where control entered must be assigned a value. Arguments do not retain their values between calls. (The ancient trick of calling one entry point with a large number of arguments to cause the procedure to "remember" the locations of those arguments, then invoking an entry with just a few arguments for later calculation, is still illegal. Furthermore, the trick doesn't work in our implementation, since arguments are not kept in static storage.)

2.4. DO Loops

do variables and range parameters may now be of integer, real, or double precision types. (The use of floating point **do** variables is very dangerous because of the possibility of unexpected roundoff, and we strongly recommend against their use.) The action of the **do** statement is now defined for all values of the **do** parameters. The statement

```
do 10 i = l, u, d
```

performs $\max(0, \lfloor (u-l+d)/d \rfloor)$ iterations. The **do** variable has a predictable value when exiting a loop: the value at the time a **goto** or **return** terminates the loop; otherwise the value that failed the limit test.

2.5. Alternate Returns

In a **subroutine** or subroutine **entry** statement, some of the arguments may be noted by an asterisk, as in

```
subroutine s(a, *, b, *)
```

The meaning of the ''alternate returns'' is described in section 5.2 of Appendix A.

3. Declarations

3.1. CHARACTER Data Type

One of the biggest improvements to the language is the addition of a character-string data type. Local and common character variables must have a length denoted by a constant expression:

```
character*17 a, b(3,4)
character*(6+3) c
```

If the length is omitted entirely, it is assumed equal to 1. A character string argument may have a constant length, or the length may be declared to be the same as that of the corresponding actual argument at run time by a statement like

```
character*(*) a
```

(There is an intrinsic function **len** that returns the actual length of a character string.) Character arrays and common blocks containing character variables must be packed: in an array of character variables, the first character of one element must follow the last character of the preceding element, without holes.

3.2. IMPLICIT Statement

The traditional implied declaration rules still hold: a variable whose name begins with **i, j, k, l, m,** or **n** is of type **integer**; other variables are of type **real,** unless otherwise declared. This general rule may be overridden with an **implicit** statement:

```
implicit real(a-c,g), complex(w-z), character*(17) (s)
```

declares that variables whose name begins with an **a ,b, c,** or **g** are **real,** those beginning with **w, x, y,** or **z** are assumed **complex,** and so on. It is still poor practice to depend on implicit typing, but this statement is an industry standard.

3.3. PARAMETER Statement

It is now possible to give a constant a symbolic name, as in

```
character str*(*)
parameter (x=17, y=x/3, pi=3.14159d0, str='hello')
```

The type of each parameter name is governed by the same implicit and explicit rules as for a variable. Symbolic names for **character** constants may be declared with an implied length ''(*)''. The right side of each equal sign must be a constant expression (an expression made up of constants, operators, and already defined parameters).

3.4. Array Declarations

Arrays may now have as many as seven dimensions. (Only three were permitted in 1966.) The lower bound of each dimension may be declared to be other than 1 by using a colon. Furthermore, an adjustable array bound may be an integer expression involving constants, arguments, and variables in **common.**

```
real a(–5:3, 7, m:n), b(n+1:2*n)
```

The upper bound on the last dimension of an array argument may be denoted by an asterisk to indicate that the upper bound is not specified:

```
integer a(5, *),  b(*), c(0:1, –2:*)
```

3.5. SAVE Statement

A little known rule of Fortran 66 is that variables in a procedure do not necessarily retain their values between invocations of that procedure. This rule permits overlay and stack implementations for the affected variables. In Fortran 77, three types of variables automatically keep there values: variables in blank common, variables defined in **data** statements and never changed, and variables in named common blocks which have not become undefined. At any instant in the execution of a program, if a named common block is declared neither in the currently executing procedure nor in any of the procedures in the chain of callers, all of the variables in that common block become undefined. Fortran 77 permits one to specify that certain variables and common blocks are to retain their values between invocations. The declaration

```
save a, /b/, c
```

leaves the values of the variables **a** and **c** and all of the contents of common block **b** unaffected by an exit from the procedure. The simple declaration

```
save
```

has this effect on all variables and common blocks in the procedure. A common block must be **save**d in every procedure in which it is declared if the desired effect is to occur.

3.6. INTRINSIC Statement

All of the functions specified in the Standard are in a single category, ‘‘intrinsic functions’’, rather than being divided into ‘‘intrinsic’’ and ‘‘basic external’’ functions. If an intrinsic function is to be passed to another procedure, it must be declared **intrinsic.** Declaring it **external** (as in Fortran 66) causes a function other than the built-in one to be passed.

4. Expressions

4.1. Character Constants

Character string constants are marked by strings surrounded by apostrophes. If an apostrophe is to be included in a constant, it is repeated:

```
′abc′
′ain′′t′
```

Although null (zero-length) character strings are not allowed in the standard Fortran, they may be used with **f77**. Our compiler has two different quotation marks, “ ′ ” and “ " ”. (See section 2.9 in the main text.)

4.2. Concatenation

One new operator has been added, character string concatenation, marked by a double slash ‘‘//’’. The result of a concatenation is the string containing the characters of the left operand followed by the charac-

ters of the right operand. The character expressions

```
'ab' // 'cd'
'abcd'
```

are equal.

Dummy arguments of type character may be declared with implied lengths:

```
subroutine s ( a, b )
character a*(*), b*(*)
```

Such dummy arguments may be used in concatenations in assign statements:

```
s = a // b
```

but not in other contexts. For example:

```
if( a // b .eq. 'abc' ) key = 1
call sub( a // b )
```

are legal statements if "a" and "b" are dummy arguments declared with explicit lengths, or if they are not arguments. These are illegal if they are declared with implied lengths.

4.3. Character String Assignment

The left and right sides of a character assignment may not share storage. (The assumed implementation of character assignment is to copy characters from the right to the left side.) If the left side is longer than the right, it is padded with blanks. If the left side is shorter than the right, trailing characters are discarded. Since the two sides of a character assignment must be disjoint, the following are illegal:

```
str = ' ' // str
str = str(2:)
```

These are not flagged as errors during compilation or execution, however the result is undefined.

4.4. Substrings

It is possible to extract a substring of a character variable or character array element, using the colon notation:

```
a(i,j) (m:n)
```

is the string of $(n-m+1)$ characters beginning at the m^{th} character of the character array element a_{ij}. Results are undefined unless $m \leq n$. Substrings may be used on the left sides of assignments and as procedure actual arguments.

4.5. Exponentiation

It is now permissible to raise real quantities to complex powers, or complex quantities to real or complex powers. (The principal part of the logarithm is used.) Also, multiple exponentiation is now defined:

```
a**b**c is equivalent to a ** (b**c)
```

4.6. Relaxation of Restrictions

Mixed mode expressions are now permitted. (For instance, it is permissible to combine integer and complex quantities in an expression.)

Constant expressions are permitted where a constant is allowed, except in **data** statements and **format** statements. (A constant expression is made up of explicit constants and **parameter**s and the Fortran operators, except for exponentiation to a floating-point power.) An adjustable dimension may now be an integer expression involving constants, arguments, and variables in **common.**

Subscripts may now be general integer expressions; the old $cv \pm c'$ rules have been removed. **do** loop bounds may be general integer, real, or double precision expressions. Computed **goto** expressions and I/O unit numbers may be general integer expressions.

5. Executable Statements

5.1. IF-THEN-ELSE

At last, the if-then-else branching structure has been added to Fortran. It is called a "Block If". A Block If begins with a statement of the form

```
if ( . . . ) then
```

and ends with an

```
end if
```

statement. Two other new statements may appear in a Block If. There may be several

```
else if (. . .) then
```

statements, followed by at most one

```
else
```

statement. If the logical expression in the Block If statement is true, the statements following it up to the next **else if**, **else,** or **end if** are executed. Otherwise, the next **else if** statement in the group is executed. If none of the **else if** conditions are true, control passes to the statements following the **else** statement, if any. (The **else** block must follow all **else if** blocks in a Block If. Of course, there may be Block Ifs embedded inside of other Block If structures.) A case construct may be rendered:

```
if (s .eq. 'ab') then
 . . .
else if (s .eq. 'cd') then
 . . .
else
 . . .
end if
```

5.2. Alternate Returns

Some of the arguments of a subroutine call may be statement labels preceded by an asterisk, as in:

```
call joe(j, *10, m, *2)
```

A **return** statement may have an integer expression, such as:

```
return k
```

If the entry point has n alternate return (asterisk) arguments and if $1 \le k \le n$, the return is followed by a branch to the corresponding statement label; otherwise the usual return to the statement following the **call** is executed.

6. Input/Output

6.1. Format Variables

A format may be the value of a character expression (constant or otherwise), or be stored in a character array, as in:

```
write(6, '(i5)') x
```

6.2. END=, ERR=, and IOSTAT= Clauses

A **read** or **write** statement may contain **end=, err=,** and **iostat=** clauses, as in:

```
write(6, 101, err=20, iostat=a(4))
read(5, 101, err=20, end=30, iostat=x)
```

Here 5 and 6 are the *units* on which the I/O is done, 101 is the statement number of the associated format, 20 and 30 are statement numbers, and **a** and **x** are integer variables. If an error occurs during I/O, control returns to the program at statement 20. If the end of the file is reached, control returns to the program at statement 30. In any case, the variable referred to in the **iostat=** clause is given a value when the I/O statement finishes. (Yes, the value is assigned to the name on the right side of the equal sign.) This value is zero if all went well, negative for end of file, and some positive value for errors.

6.3. Formatted I/O

6.3.1. Character Constants

Character constants in formats are copied literally to the output.

A format may be specified as a character constant within the **read** or **write** statement.

```
write(6,'(i2,'' isn''''t '',i1)') 7, 4
```

produces

```
 7 isn't 4
```

In the example above, the format is the character constant

```
(i2,' isn''t ',i1)
```

and the embedded character constant

```
 isn't
```

is copied into the output.

The example could have been written more legibly by taking advantage of the two types of quote marks.

```
write(6,'(i2," isn''t ",i1)') 7, 4
```

However, the double quote is not standard Fortran 77.

The standard does not allow reading into character constants or Hollerith fields. In order to facilitate running older programs, the Fortran I/O library allows reading into Hollerith fields; however this is a practice to be avoided.

6.3.2. Positional Editing Codes

t, tl, tr, and **x** codes control where the next character is in the record. **tr***n* or *n***x** specifies that the next character is *n* to the right of the current position. **tl***n* specifies that the next character is *n* to the left of the current position, allowing parts of the record to be reconsidered. **t***n* says that the next character is to be character number *n* in the record. (See section 3.3 in the main text.)

6.3.3. Colon

A colon in the format terminates the I/O operation if there are no more data items in the I/O list, otherwise it has no effect. In the fragment

```
x='("hello", :, " there", i4)'
write(6, x) 12
write(6, x)
```

the first **write** statement prints

```
hello there 12
```

while the second only prints

```
hello
```

6.3.4. Optional Plus Signs

According to the Standard, each implementation has the option of putting plus signs in front of non-negative numeric output. The **sp** format code may be used to make the optional plus signs actually appear for all subsequent items while the format is active. The **ss** format code guarantees that the I/O system will not insert the optional plus signs, and the **s** format code restores the default behavior of the I/O system. (Since we never put out optional plus signs, **ss** and **s** codes have the same effect in our implementation.)

6.3.5. Blanks on Input

Blanks in numeric input fields, other than leading blanks, will be ignored following a **bn** code in a format statement, and will be treated as zeros following a **bz** code in a format statement. The default for a unit may be changed by using the **open** statement. (Blanks are ignored by default.)

6.3.6. Unrepresentable Values

The Standard requires that if a numeric item cannot be represented in the form required by a format code, the output field must be filled with asterisks. (We think this should have been an option.)

6.3.7. Iw.m

There is a new integer output code, **i***w.m*. It is the same as **i***w*, except that there will be at least *m* digits in the output field, including, if necessary, leading zeros. The case **i***w*.0 is special, in that if the value being printed is 0, the output field is entirely blank. **i***w***.1** is the same as **i***w*.

6.3.8. Floating Point

On input, exponents may start with the letter **E, D, e,** or **d.** All have the same meaning. On output we always use **e** or **d**. The **e** and **d** format codes also have identical meanings. A leading zero before the decimal point in **e** output without a scale factor is optional with the implementation. There is a **g***w.d* format code which is the same as **e***w.d* and **f***w.d* on input, but which chooses **f** or **e** formats for output depending on the size of the number and of *d*.

6.3.9. "A" Format Code

The **a** code is used for character data. **a***w* uses a field width of *w*, while a plain **a** uses the length of the internal character item.

6.4. Standard Units

There are default formatted input and output units. The statement

```
read 10, a, b
```

reads from the standard unit using format statement 10. The default unit may be explicitly specified by an asterisk, as in

```
read(*, 10) a, b
```

Similarly, the standard output unit is specified by a **print** statement or an asterisk unit:

```
print 10
write(*, 10)
```

6.5. List-Directed I/O

List-directed I/O is a kind of free form input for sequential I/O. It is invoked by using an asterisk as the format identifier, as in

```
read(6, *) a,b,c
```

On input, values are separated by strings of blanks and possibly a comma. On UNIX, tabs may be used interchangeably with blanks as separators. Values, except for character strings, cannot contain blanks. End of record counts as a blank, except in character strings, where it is ignored. Complex constants are given as two real constants separated by a comma and enclosed in parentheses. A null input field, such as between two consecutive commas, means the corresponding variable in the I/O list is not changed. Values may be preceded by repetition counts, as in

```
4*(3.,2.)  2*, 4*'hello'
```

which stands for 4 complex constants, 2 null values, and 4 string constants.

The Fortran standard requires data being read into **character** variables by a list-directed read to be enclosed in quotes. In our system, the quotes are optional for strings which do not start with a digit or quote and do not contain separators.

For output, suitable formats are chosen for each item. The values of character strings are printed; they are not enclosed in quotes. According to the standard, they could not be read back using list-directed input. However much of this data could be read back in with list-directed I/O on our system.

6.6. Direct I/O

A file connected for direct access consists of a set of equal-sized records each of which is uniquely identified by a positive integer. The records may be written or read in any order, using direct access I/O statements.

Direct access **read** and **write** statements have an extra argument, **rec=,** which gives the record number to be read or written.

```
read(2, rec=13, err=20) (a(i), i=1, 203)
```

reads the thirteenth record into the array **a.**

The size of the records must be given by an **open** statement (see below). Direct access files may be connected for either formatted or unformatted I/O.

6.7. Internal Files

Internal files are character string objects, such as variables or substrings, or arrays of type character. In the former cases there is only a single record in the file; in the latter case each array element is a record. The Standard includes only sequential formatted I/O on internal files. (I/O is not a very precise term to use here, but internal files are dealt with using **read** and **write.)** Internal files are used by giving the name of the character object in place of the unit number, as in

```
character*80 x
read(5,'(a)') x
read(x,'(i3,i4)') n1,n2
```

which reads a character string into **x** and then reads two integers from the front of it. A sequential **read** or **write** always starts at the beginning of an internal file.

We also support two extensions of the standard. The first is direct I/O on internal files. This is like direct I/O on external files, except that the number of records in the file cannot be changed. In this case a record is a single element of an array of character strings. The second extension is list-directed I/O on internal files.

6.8. OPEN, CLOSE, and INQUIRE Statements

These statements are used to connect and disconnect units and files, and to gather information about units and files.

6.8.1. OPEN

The **open** statement is used to connect a file with a unit, or to alter some properties of the connection. The following is a minimal example.

```
open(1, file='fort.junk')
```

open takes a variety of arguments with meanings described below.

unit= an integer between 0 and 99 inclusive which is the unit to which the file is to be connected (see section 5.3 in the text). If this parameter is the first one in the **open** statement, the **unit=** can be omitted.

iostat= is the same as in **read** or **write.**

err= is the same as in **read** or **write.**

file= a character expression, which when stripped of trailing blanks, is the name of the file to be connected to the unit. The file name should not be given if the **status='scratch'.**

status= one of **'old', 'new', 'scratch',** or **'unknown'.** If this parameter is not given, **'unknown'** is assumed. The meaning of **'unknown'** is processor dependent; our system will create the file if it doesn't exist. If **'scratch'** is given, a temporary file will be created. Temporary files are destroyed at the end of execution. If **'new'** is given, the file must not exist. It will be created for both reading and writing. If **'old'** is given, it is an error for the file not to exist.

access= 'sequential' or **'direct',** depending on whether the file is to be opened for sequential or direct I/O.

form= 'formatted' or **'unformatted'.** On UNIX systems, **form='print'** implies **'formatted'** with vertical format control. (See section 3.4 of the text).

recl= a positive integer specifying the record length of the direct access file being opened. We measure all record lengths in bytes. On UNIX systems a record length of 1 has the special meaning explained in section 5.1 of the text.

blank= 'null' or **'zero'.** This parameter has meaning only for formatted I/O. The default value is **'null'.** **'zero'** means that blanks, other than leading blanks, in numeric input fields are to be treated as zeros.

Opening a new file on a unit which is already connected has the effect of first closing the old file.

6.8.2. CLOSE

close severs the connection between a unit and a file. The unit number must be given. The optional parameters are **iostat=** and **err=** with their usual meanings, and **status=** either **'keep'** or **'delete'.** For scratch files the default is **'delete';** otherwise **'keep'** is the default. **'delete'** means the file will be removed. A simple example is

```
close(3, err=17)
```

6.8.3. INQUIRE

The **inquire** statement gives information about a unit (''inquire by unit'') or a file (''inquire by file''). Simple examples are:

```
inquire(unit=3, name=xx)
inquire(file='junk', number=n, exist=l)
```

file= a character variable specifies the file the **inquire** is about. Trailing blanks in the file name are ignored.

unit= an integer variable specifies the unit the **inquire** is about. Exactly one of **file=** or **unit=** must be used.

iostat=, err= are as before.

exist= a logical variable. The logical variable is set to **.true.** if the file or unit exists and is set to **.false.** otherwise.

opened= a logical variable. The logical variable is set to **.true.** if the file is connected to a unit or if the unit is connected to a file, and it is set to **.false.** otherwise.

number= an integer variable to which is assigned the number of the unit connected to the file, if any.

named= a logical variable to which is assigned **.true.** if the file has a name, or **.false.** otherwise.

name= a character variable to which is assigned the name of the file (inquire by file) or the name of the file connected to the unit (inquire by unit).

access= a character variable to which will be assigned the value **'sequential'** if the connection is for sequential I/O, **'direct'** if the connection is for direct I/O, **'unknown'** if not connected.

sequential= a character variable to which is assigned the value **'yes'** if the file could be connected for sequential I/O, **'no'** if the file could not be connected for sequential I/O, and **'unknown'** if we can't tell.

direct= a character variable to which is assigned the value **'yes'** if the file could be connected for direct I/O, **'no'** if the file could not be connected for direct I/O, and **'unknown'** if we can't tell.

form= a character variable to which is assigned the value **'unformatted'** if the file is connected for unformatted I/O, **'formatted'** if the file is connected for formatted I/O, **'print'** for formatted I/O with vertical format control, or **'unknown'** if not connected.

formatted= a character variable to which is assigned the value **'yes'** if the file could be connected for formatted I/O, **'no'** if the file could not be connected for formatted I/O, and **'unknown'** if we can't tell.

unformatted= a character variable to which is assigned the value **'yes'** if the file could be connected for unformatted I/O, **'no'** if the file could not be connected for unformatted I/O, and **'unknown'** if we can't tell.

recl= an integer variable to which is assigned the record length of the records in the file if the file is connected for direct access.

nextrec= an integer variable to which is assigned one more than the number of the the last record read from a file connected for direct access.

blank= a character variable to which is assigned the value **'null'** if null blank control is in effect for the file connected for formatted I/O, **'zero'** if blanks are being converted to zeros and the file is connected for formatted I/O.

For information on file permissions, ownership, etc., use the Fortran library routines **stat** and **access**.

For further discussion of the UNIX Fortran I/O system see ''Introduction to the f77 I/O Library'' [9].

APPENDIX B: References and Bibliography

References

1. *American National Standard Programming Language FORTRAN, ANSI X3.9-1978*. New York: American National Standards Institute, 1978.
2. *USA Standard FORTRAN, USAS X3.9-1966*. New York: United States of America Standards Institute, 1966. Clarified in *Comm. ACM* 12:289 (1969) and *Comm. ACM* 14:628 (1971).
3. Kernighan, B. W., and D. M. Ritchie. *The C Programming Language.* Englewood Cliffs: Prentice-Hall, 1978.
4. Ritchie, D. M. Private communication.
5. Johnson, S. C. ''A Portable Compiler: Theory and Practice,'' *Proceedings of Fifth ACM Symposium on Principles of Programming Languages*. 1978.
6. Feldman, S. I. ''An Informal Description of EFL,'' internal memorandum.
7. Kernighan, B. W. ''RATFOR—A Preprocessor for Rational Fortran,'' *Bell Laboratories Computing Science Technical Report #55*. 1977.
8. Ritchie, D. M. Private communication.
9. Wasley, D. L. ''Introduction to the f77 I/O Library'', *UNIX Programmer's Manual, Volume 2c*.

Bibliography

The following books or documents describe aspects of Fortran 77. This list cannot pretend to be complete. Certainly no particular endorsement is implied.

1. Brainerd, Walter S., et al. *Fortran 77 Programming.* Harper Row, 1978.
2. Day, A. C. *Compatible Fortran.* Cambridge University Press, 1979.
3. Dock, V. Thomas. *Structured Fortran IV Programming.* West, 1979.
4. Feldman, S. I. ''The Programming Language EFL,'' *Bell Laboratories Technical Report*. June 1979.
5. Hume, J. N., and R. C. Holt. *Programming Fortran 77.* Reston, 1979.
6. Katzan, Harry, Jr. *Fortran 77.* Van Nostrand-Reinhold, 1978.
7. Meissner, Loren P., and Organick, Elliott I. *Fortran 77 Featuring Structured Programming*, Addison-Wesley, 1979.
8. Merchant, Michael J. *ABC's of Fortran Programming.* Wadsworth, 1979.
9. Page, Rex, and Richard Didday. *Fortran 77 for Humans.* West, 1980.
10. Wagener, Jerrold L. *Principles of Fortran 77 Programming.* Wiley, 1980.

Table of Contents

1. Introduction 4
1.1. Usage 4
1.2. Documentation Conventions 6
1.3. Implementation Strategy 6
1.4. Debugging Aids 6
2. Language Extensions 6
2.1. Double Complex Data Type 6
2.2. Internal Files 6
2.3. Implicit Undefined Statement 7
2.4. Recursion 7
2.5. Automatic Storage 7
2.6. Source Input Format 7
2.7. Include Statement 7
2.8. Binary Initialization Constants 8
2.9. Character Strings 8
2.10. Hollerith 8
2.11. Equivalence Statements 8
2.12. One-Trip DO Loops 8
2.13. Commas in Formatted Input 9
2.14. Short Integers 9
2.15. Additional Intrinsic Functions 9
2.16. Namelist I/O 9
2.17. Automatic Precision Increase 11
2.18. Characters and Integers 12
3. Violations of the Standard 12
3.1. Double Precision Alignment 12
3.2. Dummy Procedure Arguments 12
3.3. T and TL Formats 12
3.4. Carriage Control 12
3.5. Assigned Goto 12
4. Inter-Procedure Interface 13
4.1. Procedure Names 13
4.2. Data Representations 13
4.3. Arrays 13
4.4. Return Values 13
4.5. Argument Lists 14
4.6. System Interface 14
5. File Formats 15
5.1. Structure of Fortran Files 15
5.2. Portability Considerations 15
5.3. Logical Units and Files 15
Appendix A. Differences Between Fortran 66 and Fortran 77 16
1. Features Deleted from Fortran 66 16
1.1. Hollerith 16
1.2. Extended Range of DO 16
2. Program Form 16
2.1. Blank Lines 16

2.2. Program and Block Data Statements 16
2.3. ENTRY Statement 16
2.4. DO Loops 17
2.5. Alternate Returns 17
3. Declarations 17
3.1. CHARACTER Data Type 17
3.2. IMPLICIT Statement 17
3.3. PARAMETER Statement 17
3.4. Array Declarations 18
3.5. SAVE Statement 18
3.6. INTRINSIC Statement 18
4. Expressions 18
4.1. Character Constants 18
4.2. Concatenation 18
4.3. Character String Assignment 19
4.4. Substrings 19
4.5. Exponentiation 19
4.6. Relaxation of Restrictions 19
5. Executable Statements 20
5.1. IF-THEN-ELSE 20
5.2. Alternate Returns 20
6. Input/Output 20
6.1. Format Variables 20
6.2. END=, ERR=, and IOSTAT= Clauses 21
6.3. Formatted I/O 21
6.4. Standard Units 22
6.5. List-Directed I/O 23
6.6. Direct I/O 23
6.7. Internal Files 23
6.8. OPEN, CLOSE, and INQUIRE Statements 24
Appendix B. References and Bibliography 26

Introduction to the f77 I/O Library

David L. Wasley

J. Berkman

University of California, Berkeley
Berkeley, California 94720

ABSTRACT

The f77 I/O library, libI77.a, includes routines to perform all of the standard types of Fortran input and output specified in the ANSI 1978 Fortran standard. The I/O Library was written originally by Peter J. Weinberger at Bell Labs. Where the original implementation was incomplete, it has been rewritten to more closely implement the standard. Where the standard is vague, we have tried to provide flexibility within the constraints of the UNIX† operating system. A number of logical extensions and enhancements have been provided such as the use of the C stdio library routines to provide efficient buffering for file I/O.

Revised September, 1985

† UNIX is a registered trademark of The Open Group in the U.S. and other countries.

1. Fortran I/O

The requirements of the ANSI standard impose significant overhead on programs that do large amounts of I/O. Formatted I/O can be very ‘‘expensive’’ while direct access binary I/O is usually very efficient. Because of the complexity of Fortran I/O, some general concepts deserve clarification.

1.1. Types of I/O and logical records

There are four forms of I/O: **formatted, unformatted, list directed,** and **namelist.** The last two are related to formatted but do not obey all the rules for formatted I/O. There are two types of ‘‘files’’: **external** and **internal** and two modes of access to files: **direct** and **sequential.** The definition of a logical record depends upon the combination of I/O form, file type, and access mode specified by the Fortran I/O statement.

1.1.1. Direct access external I/O

A logical record in a **direct** access **external** file is a string of bytes of a length specified when the file is opened. Read and write statements must not specify logical records longer than the original record size definition. Shorter logical records are allowed. **Unformatted** direct writes leave the unfilled part of the record undefined. **Formatted** direct writes cause the unfilled record to be padded with blanks.

1.1.2. Sequential access external I/O

Logical records in **sequentially** accessed **external** files may be of arbitrary and variable length. Logical record length for **unformatted** sequential files is determined by the size of items in the iolist. The requirements of this form of I/O cause the external physical record size to be somewhat larger than the logical record size. For **formatted** write statements, logical record length is determined by the format statement interacting with the iolist at execution time. The ‘‘newline’’ character is the logical record delimiter. Formatted sequential access causes one or more logical records ending with ‘‘newline’’ characters to be read or written.

1.1.3. List directed and namelist sequential external I/O

Logical record length for **list directed** and **namelist** I/O is relatively meaningless. On output, the record length is dependent on the magnitude of the data items. On input, the record length is determined by the data types and the file contents. By ANSI definition, a slash, ‘‘/’’, terminates execution of a list directed input operation. Namelist input is terminated by ‘‘&end’’ or ‘‘$end’’ (depending on whether the character before the namelist name was ‘‘&’’ or ‘‘$’’).

1.1.4. Internal I/O

The logical record length for an **internal** read or write is the length of the character variable or array element. Thus a simple character variable is a single logical record. A character variable array is similar to a fixed length direct access file, and obeys the same rules. **Unformatted** and **namelist** I/O are not allowed on ‘‘internal’’ files.

1.2. I/O execution

Note that each execution of a Fortran **unformatted** I/O statement causes a single logical record to be read or written. Each execution of a Fortran **formatted** I/O statement causes one or more logical records to be read or written.

A slash, ‘‘/’’, will terminate assignment of values to the input list during **list directed** input and the remainder of the current input line is skipped. The standard is rather vague on this point but seems to require that a new external logical record be found at the start of any formatted input. Therefore data following the slash is ignored and may be used to comment the data file.

Direct access list directed I/O is not allowed. **Unformatted internal** I/O is not allowed. **Namelist** I/O is allowed only with **external sequential** files. All other flavors of I/O are allowed, although some are not part of the ANSI standard.

Any I/O statement may include an **err=** clause to specify an alternative branch to be taken on errors and/or an **iostat=** clause to return the specific error code. Any error detected during I/O processing will cause the program to abort unless either **err=** or **iostat=** has been specificed in the program. Read statements may include **end=** to branch on end-of-file. The end-of-file indication for that logical unit may be reset with a **backspace** statement. File position and the value of I/O list items is undefined following an error.

2. Implementation details

Some details of the current implementation may be useful in understanding constraints on Fortran I/O.

2.1. Number of logical units

Unit numbers must be in the range 0 – 99. The maximum number of logical units that a program may have open at one time is the same as the UNIX system limit, currently 48.

2.2. Standard logical units

By default, logical units 0, 5, and 6 are opened to ''stderr'', ''stdin'', and ''stdout'' respectively. However they can be re-defined with an **open** statement. To preserve error reporting, it is an error to close logical unit 0 although it may be reopened to another file.

If you want to open the default file name for any preconnected logical unit, remember to **close** the unit first. Redefining the standard units may impair normal console I/O. An alternative is to use shell re-direction to externally re-define the above units. To re-define default blank control or format of the standard input or output files, use the **open** statement specifying the unit number and no file name (see § 2.4).

The standard units, 0, 5, and 6, are named internally ''stderr'', ''stdin'', and ''stdout'' respectively. These are not actual file names and can not be used for opening these units. **Inquire** will not return these names and will indicate that the above units are not named unless they have been opened to real files. The names are meant to make error reporting more meaningful.

2.3. Vertical format control

Simple vertical format control is implemented. The logical unit must be opened for sequential access with **form = 'print'** (see § 3.2). Control codes ''0'' and ''1'' are replaced in the output file with ''\n'' and ''\f'' respectively. The control character ''+'' is not implemented and, like any other character in the first position of a record written to a ''print'' file, is dropped. The **form = 'print'** mode does not recognize vertical format control for **direct formatted, list directed,** or **namelist** output.

An alternative is to use the filter *fpr*(1) for vertical format control. It replaces ''0'' and ''1'' by ''\n'' and ''\f'' respectively, and implements the ''+'' control code. Unlike **form = 'print'** which drops unrecognized form control characters, *fpr* copies those characters to the output file.

2.4. File names and the open statement

A file name may be specified in an **open** statement for the logical unit. If a logical unit is opened by an **open** statement which does not specify a file name, or it is opened implicitly by the execution of a **read**, **write**, or **endfile** statement, then the default file name is *fort.N* where *N* is the logical unit number. Before opening the file, the library checks for an environment variable with a name identical to the tail of the file name with periods removed.† If it finds such an environment variable, it uses its value as the actual name of the file. For example, a program containing:

†Periods are deleted because they can not be part of environment variable names in the Bourne shell.

```
open(32,file='/usr/guest/census/data.d')
read(32,100) vec
write(44) vec
```

normally will read from */usr/guest/census/data.d* and write to *fort.44* in the current directory. If the environment variables *datad* and *fort44* are set, e.g.:

```
% setenv datad mydata
% setenv fort44 myout
```

in the C shell or:

```
$ datad=mydata
$ fort44=myout
$ export datad fort44
```

in the Bourne shell, then the program will read from *mydata* and write to *myout*.

An **open** statement need not specify a file name. If it refers to a logical unit that is already open, the **blank=** and **form=** specifiers may be redefined without affecting the current file position. Otherwise, if **status = 'scratch'** is specified, a temporary file with a name of the form *tmp.FXXXXXX* will be opened, and, by default, will be deleted when closed or during termination of program execution.

It is an error to try to open an existing file with **status = 'new'** . It is an error to try to open a nonexistent file with **status = 'old'** . By default, **status = 'unknown'** will be assumed, and a file will be created if necessary.

By default, files are positioned at their beginning upon opening, but see *fseek*(3f) and *ioinit*(3f) for alternatives. Existing files are never truncated on opening. Sequentially accessed external files are truncated to the current file position on **close**, **backspace**, or **rewind** only if the last access to the file was a write. An **endfile** always causes such files to be truncated to the current file position.

2.5. Format interpretation

Formats which are in format statements are parsed by the compiler; formats in **read**, **write**, and **print** statements are parsed during execution by the I/O library. Upper as well as lower case characters are recognized in format statements and all the alphabetic arguments to the I/O library routines.

If the external representation of a datum is too large for the field width specified, the specified field is filled with asterisks (*). On **E**w.d**E**e output, the exponent field will be filled with asterisks if the exponent representation is too large. This will only happen if "e" is zero (see appendix B).

On output, a real value that is truly zero will display as "0." to distinguish it from a very small non-zero value. If this causes problems for other input systems, the **BZ** edit descriptor may be used to cause the field following the decimal point to be filled with zero's.

Non-destructive tabbing is implemented for both internal and external formatted I/O. Tabbing left or right on output does not affect previously written portions of a record. Tabbing right on output causes unwritten portions of a record to be filled with blanks. Tabbing right off the end of an input logical record is an error. Tabbing left beyond the beginning of an input logical record leaves the input pointer at the beginning of the record. The format specifier **T** must be followed by a positive non-zero number. If it is not, it will have a different meaning (see § 3.1).

Tabbing left requires seek ability on the logical unit. Therefore it is not allowed in I/O to a terminal or pipe. Likewise, nondestructive tabbing in either direction is possible only on a unit that can seek. Otherwise tabbing right or spacing with **X** will write blanks on the output.

2.6. List directed output

In formatting list directed output, the I/O system tries to prevent output lines longer than 80 characters. Each external datum will be separated by two spaces. List directed output of **complex** values includes an appropriate comma. List directed output distinguishes between **real** and **double precision**

values and formats them differently. Output of a character string that includes "\n" is interpreted reasonably by the output system.

2.7. I/O errors

If I/O errors are not trapped by the user's program an appropriate error message will be written to "stderr" before aborting. An error number will be printed in "[ ]" along with a brief error message showing the logical unit and I/O state. Error numbers < 100 refer to UNIX errors, and are described in the introduction to chapter 2 of the UNIX Programmer's Manual. Error numbers ≥ 100 come from the I/O library, and are described further in the appendix to this writeup‡. For internal I/O, part of the string will be printed with "|" at the current position in the string. For external I/O, part of the current record will be displayed if the error was caused during reading from a file that can backspace.

3. Non-"ANSI Standard" extensions

Several extensions have been added to the I/O system to provide for functions omitted or poorly defined in the standard. Programmers should be aware that these are non-portable.

3.1. Format specifiers

B is an acceptable edit control specifier. It causes return to the logical unit's default mode of blank interpretation. This is consistent with **S** which returns to default sign control.

P by itself is equivalent to **0P** . It resets the scale factor to the default value, 0.

The form of the **E**w.d**E**e format specifier has been extended to **D** also. The form **E**w.d.e is allowed but is not standard. The "e" field specifies the minimum number of digits or spaces in the exponent field on output. If the value of the exponent is too large, the exponent notation **e** or **d** will be dropped from the output to allow one more character position. If this is still not adequate, the "e" field will be filled with asterisks (*). The default value for "e" is 2.

An additional form of tab control specification has been added. The ANSI standard forms **TR**n, **TL**n, and **T**n are supported where *n* is a positive non-zero number. If **T** or n**T** is specified, tabbing will be to the next (or n-th) 8-column tab stop. Thus columns of alphanumerics can be lined up without counting.

A format control specifier has been added to suppress the newline at the end of the last record of a formatted sequential write. The specifier is a dollar sign ($). It is constrained by the same rules as the colon (:). It is used typically for console prompts. For example:

```
write (*, "('enter value for x: ',$)")
read (*,*) x
```

Radices other than 10 can be specified for formatted integer I/O conversion. The specifier is patterned after **P,** the scale factor for floating point conversion. It remains in effect until another radix is specified or format interpretation is complete. The specifier is defined as [n]**R** where $2 \le n \le 36$. If *n* is omitted, the default decimal radix is restored.

The format specifier **O**m.n may be used for an octal conversion; it is equivalent to 8**R,I**m.n,10**R**. Similarly, **Z**m.n is equivalent to 16**R,I**m.n,10**R** and may be used for an hexadecimal conversion;

In conjunction with the above, a sign control specifier has been added to cause integer values to be interpreted as unsigned during output conversion. The specifier is **SU** and remains in effect until

---

‡ On many systems, these are also available in *help f77 io_err_msgs*.

†Note: Unsigned integer values greater than (2**31 - 1), can be read and written using **SU**. However they can not be used in computations because Fortran uses signed arithmetic and such values appear to the arithmetic unit as negative numbers.

another sign control specifier is encountered, or format interpretation is complete.† Radix and ''unsigned'' specifiers could be used to format a hexadecimal dump, as follows:

```
2000  format ( SU, 8Z10.8 )
```

3.2. Print files

The ANSI standard is ambiguous regarding the definition of a ''print'' file. Since UNIX has no default ''print'' file, an additional **form=** specifier is now recognized in the **open** statement. Specifying **form = ′print′** implies **formatted** and enables vertical format control for that logical unit (see § 2.3). Vertical format control is interpreted only on sequential formatted writes to a ''print'' file.

The **inquire** statement will return **print** in the **form=** string variable for logical units opened as ''print'' files. It will return -1 for the unit number of an unconnected file.

If a logical unit is already open, an **open** statement including the **form=** option or the **blank=** option will do nothing but re-define those options. This instance of the **open** statement need not include the file name, and must not include a file name if **unit=** refers to a standard input or output. Therefore, to re-define the standard output as a ''print'' file, use:

```
open (unit=6, form='print')
```

3.3. Scratch files

A **close** statement with **status = ′keep′** may be specified for temporary files. This is the default for all other files. Remember to get the scratch file's real name, using **inquire** , if you want to re-open it later.

3.4. List directed I/O

List directed read has been modified to allow tab characters wherever blanks are allowed. It also allows input of a string not enclosed in quotes. The string must not start with a digit or quote, and can not contain any separators ('','', ''/'', blank or tab). A newline will terminate the string unless escaped with \. Any string not meeting the above restrictions must be enclosed in quotes (" " " or " ′ ").

Internal list directed I/O has been implemented. During internal list reads, bytes are consumed until the iolist is satisfied, or the ''end-of-file'' is reached. During internal list writes, records are filled until the iolist is satisfied. The length of an internal array element should be at least 20 bytes to avoid logical record overflow when writing double precision values. Internal list read was implemented to make command line decoding easier. Internal list write should be avoided.

3.5. Namelist I/O

Namelist I/O is a common extension in Fortran systems. The **f77** version was designed to be compatible with other vendors versions; it is described in ''A Portable Fortran 77 Compiler'', by Feldman and Weinberger, August, 1985.

4. Running older programs

Traditional Fortran environments usually assume carriage control on all logical units, usually interpret blank spaces on input as ''0''s, and often provide attachment of global file names to logical units at run time. There are several routines in the I/O library to provide these functions.

4.1. Traditional unit control parameters

If a program reads and writes only units 5 and 6, then including **–lI66** in the f77 command will cause carriage control to be interpreted on output and cause blanks to be zeros on input without further modification of the program. If this is not adequate, the routine *ioinit*(3f) can be called to specify control parameters separately, including whether files should be positioned at their beginning or end upon opening.

4.2. Ioinit()

Ioinit(3f) can be used to attach logical units to specific files at run time, and to set global parameters for the I/O system. It will look for names of a user specified form in the environment and open the corresponding logical unit for **sequential formatted** I/O. Names must be of the form **PREFIX***nn* where **PREFIX** is specified in the call to *ioinit* and *nn* is the logical unit to be opened. Unit numbers < 10 must include the leading ‘‘0’’.

Ioinit should prove adequate for most programs as written. However, it is written in Fortran–77 specifically so that it may serve as an example for similar user-supplied routines. A copy may be retrieved by ‘‘ar x /usr/lib/libU77.a ioinit.f’’. See § 2.4 for another way to override program file names through environment variables.

5. Magnetic tape I/O

Because the I/O library uses stdio buffering, reading or writing magnetic tapes should be done with great caution, or avoided if possible. A set of routines has been provided to read and write arbitrary sized buffers to or from tape directly. The buffer must be a **character** object. **Internal** I/O can be used to fill or interpret the buffer. These routines do not use normal Fortran I/O processing and do not obey Fortran I/O rules. See *topen*(3f).

6. Caveat Programmer

The I/O library is extremely complex yet we believe there are few bugs left. We’ve tried to make the system as correct as possible according to the ANSI X3.9–1978 document and keep it compatible with the UNIX file system. Exceptions to the standard are noted in appendix B.

Appendix A

I/O Library Error Messages

The following error messages are generated by the I/O library. The error numbers are returned in the **iostat=** variable. Error numbers < 100 are generated by the UNIX kernel. See the introduction to chapter 2 of the UNIX Programmers Manual for their description.

100 *error in format*
See error message output for the location of the error in the format. Can be caused by more than 10 levels of nested parentheses, or an extremely long format statement.

101 *illegal unit number*
It is illegal to close logical unit 0. Unit numbers must be between 0 and 99 inclusive.

102 *formatted i/o not allowed*
The logical unit was opened for unformatted I/O.

103 *unformatted i/o not allowed*
The logical unit was opened for formatted I/O.

104 *direct i/o not allowed*
The logical unit was opened for sequential access, or the logical record length was specified as 0.

105 *sequential i/o not allowed*
The logical unit was opened for direct access I/O.

106 *can't backspace file*
The file associated with the logical unit can't seek. May be a device or a pipe.

107 *off beginning of record*
The format specified a left tab beyond the beginning of an internal input record.

108 *can't stat file*
The system can't return status information about the file. Perhaps the directory is unreadable.

109 *no * after repeat count*
Repeat counts in list directed I/O must be followed by an * with no blank spaces.

110 *off end of record*
A formatted write tried to go beyond the logical end-of-record. An unformatted read or write will also cause this.

111 *truncation failed*
The truncation of an external sequential file on **close**, **backspace**, **rewind**, or **endfile** failed.

112 *incomprehensible list input*
List input has to be just right.

113 *out of free space*
The library dynamically creates buffers for internal use. You ran out of memory for this. Your program is too big!

114 *unit not connected*
The logical unit was not open.

115 *invalid data for integer format term*
Only spaces, a leading sign and digits are allowed.

116 *invalid data for logical format term*
Legal input consists of spaces (optional), a period (optional), and then a "t", "T", "f", or "F".

117 *'new' file exists*
You tried to open an existing file with "status='new'".

118 *can't find 'old' file*
You tried to open a non-existent file with "status='old'".

119 *opening too many files or unknown system error*
Either you are trying to open too many files simultaneously or there has been an undetected system error.

120 *requires seek ability*
Direct access requires seek ability. Sequential unformatted I/O requires seek ability on the file due to the special data structure required. Tabbing left also requires seek ability.

121 *illegal argument*
Certain arguments to **open**, etc. will be checked for legitimacy. Often only non-default forms are looked for.

122 *negative repeat count*
The repeat count for list directed input must be a positive integer.

123 *illegal operation for unit*
An operation was requested for a device associated with the logical unit which was not possible. This error is returned by the tape I/O routines if attempting to read past end-of-tape, etc.

124 *invalid data for d, e, f or g format term*
Input data must be legal.

125 *illegal input for namelist*
Column one of input is ignored, the namelist name must match, the variables must be in the namelist, and the data must be of the right type.

Appendix B

Exceptions to the ANSI Standard

A few exceptions to the ANSI standard remain.

Vertical format control

The "+" carriage control specifier is not fully implemented (see § 2.3). It would be difficult to implement it correctly and still provide UNIX-like file I/O.

Furthermore, the carriage control implementation is asymmetrical. A file written with carriage control interpretation can not be read again with the same characters in column 1.

An alternative to interpreting carriage control internally is to run the output file through a "Fortran output filter" before printing. This filter could recognize a much broader range of carriage control and include terminal dependent processing. One such filter is *fpr*(1).

Default files

Files created by default use of **endfile** statements are opened for **sequential formatted** access. There is no way to redefine such a file to allow **direct** or **unformatted** access.

Lower case strings

It is not clear if the ANSI standard requires internally generated strings to be upper case or not. As currently written, the **inquire** statement will return lower case strings for any alphanumeric data.

Exponent representation on Ew.dEe output

If the field width for the exponent is too small, the standard allows dropping the exponent character but only if the exponent is > 99. This system does not enforce that restriction. Further, the standard implies that the entire field, "w", should be filled with asterisks if the exponent can not be displayed. This system fills only the exponent field in the above case since that is more diagnostic.

Pre-connection of files

The standard says units must be pre-connected to files before the program starts or must be explicitly opened. Instead, the I/O library connects the unit to a file on its first use in a **read**, **write**, **print**, or **endfile** statement. Thus **inquire** by unit can not tell prior to a unit number use the characteristics or name of the file corresponding to a unit.

Table of Contents

1. Fortran I/O 3
1.1. Types of I/O and logical records 3
1.1.1. Direct access external I/O 3
1.1.2. Sequential access external I/O 3
1.1.3. List directed and namelist sequential external I/O 3
1.1.4. Internal I/O 3
1.2. I/O execution 3
2. Implementation details 4
2.1. Number of logical units 4
2.2. Standard logical units 4
2.3. Vertical format control 4
2.4. File names and the open statement 4
2.5. Format interpretation 5
2.6. List directed output 5
2.7. I/O errors 6
3. Non-“ANSI Standard” extensions 6
3.1. Format specifiers 6
3.2. Print files 7
3.3. Scratch files 7
3.4. List directed I/O 7
3.5. Namelist I/O 7
4. Running older programs 7
4.1. Traditional unit control parameters 8
4.2. Ioinit() 8
5. Magnetic tape I/O 8
6. Caveat Programmer 8
Appendix A: I/O Library Error Messages 9
Appendix B: Exceptions to the ANSI Standard 11

A Tutorial Introduction to ADB

J. F. Maranzano

S. R. Bourne

ABSTRACT

Debugging tools generally provide a wealth of information about the inner workings of programs. These tools have been available on UNIX† to allow users to examine ‘‘core’’ files that result from aborted programs. A new debugging program, ADB, provides enhanced capabilities to examine "core" and other program files in a variety of formats, run programs with embedded breakpoints and patch files.

ADB is an indispensable but complex tool for debugging crashed systems and/or programs. This document provides an introduction to ADB with examples of its use. It explains the various formatting options, techniques for debugging C programs, examples of printing file system information and patching.

1. Introduction

ADB is a new debugging program that is available on UNIX. It provides capabilities to look at ‘‘core’’ files resulting from aborted programs, print output in a variety of formats, patch files, and run programs with embedded breakpoints. This document provides examples of the more useful features of ADB. The reader is expected to be familiar with the basic commands on UNIX with the C language, and with References 1, 2 and 3.

2. A Quick Survey

2.1. Invocation

ADB is invoked as:

adb objfile corefile

where *objfile* is an executable UNIX file and *corefile* is a core image file. Many times this will look like:

adb a.out core

or more simply:

adb

where the defaults are *a.out* and *core* respectively. The filename minus (–) means ignore this argument as in:

adb – core

ADB has requests for examining locations in either file. The **?** request examines the contents of

† UNIX is a registered trademark of The Open Group in the U.S. and other countries.

objfile, the **/** request examines the *corefile.* The general form of these requests is:

address ? format

or

address / format

2.2. Current Address

ADB maintains a current address, called dot, similar in function to the current pointer in the UNIX editor. When an address is entered, the current address is set to that location, so that:

0126?i

sets dot to octal 126 and prints the instruction at that address. The request:

.,10/d

prints 10 decimal numbers starting at dot. Dot ends up referring to the address of the last item printed. When used with the **?** or **/** requests, the current address can be advanced by typing newline; it can be decremented by typing **^**.

Addresses are represented by expressions. Expressions are made up from decimal, octal, and hexadecimal integers, and symbols from the program under test. These may be combined with the operators +, –, *, % (integer division), & (bitwise and), | (bitwise inclusive or), # (round up to the next multiple), and ~ (not). (All arithmetic within ADB is 32 bits.) When typing a symbolic address for a C program, the user can type *name* or *_name;* ADB will recognize both forms.

2.3. Formats

To print data, a user specifies a collection of letters and characters that describe the format of the printout. Formats are "remembered" in the sense that typing a request without one will cause the new printout to appear in the previous format. The following are the most commonly used format letters.

b	**one byte in octal**
c	**one byte as a character**
o	**one word in octal**
d	**one word in decimal**
f	**two words in floating point**
i	**PDP 11 instruction**
s	**a null terminated character string**
a	**the value of dot**
u	**one word as unsigned integer**
n	**print a newline**
r	**print a blank space**
^	**backup dot**

(Format letters are also available for "long" values, for example, '**D**' for long decimal, and '**F**' for double floating point.) For other formats see the ADB manual.

2.4. General Request Meanings

The general form of a request is:

address,count command modifier

which sets 'dot' to *address* and executes the command *count* times.

The following table illustrates some general ADB command meanings:

Command	Meaning
?	**Print contents from *a.out* file**
/	**Print contents from *core* file**
=	**Print value of "dot"**
:	**Breakpoint control**
$	**Miscellaneous requests**
;	**Request separator**
!	**Escape to shell**

ADB catches signals, so a user cannot use a quit signal to exit from ADB. The request $q or $Q (or cntl-D) must be used to exit from ADB.

3. Debugging C Programs

3.1. Debugging A Core Image

Consider the C program in Figure 1. The program is used to illustrate a common error made by C programmers. The object of the program is to change the lower case "t" to upper case in the string pointed to by *charp* and then write the character string to the file indicated by argument 1. The bug shown is that the character "T" is stored in the pointer *charp* instead of the string pointed to by *charp.* Executing the program produces a core file because of an out of bounds memory reference.

ADB is invoked by:

adb a.out core

The first debugging request:

$c

is used to give a C backtrace through the subroutines called. As shown in Figure 2 only one function (*main*) was called and the arguments *argc* and *argv* have octal values 02 and 0177762 respectively. Both of these values look reasonable; 02 = two arguments, 0177762 = address on stack of parameter vector.
The next request:

$C

is used to give a C backtrace plus an interpretation of all the local variables in each function and their values in octal. The value of the variable *cc* looks incorrect since *cc* was declared as a character.

The next request:

$r

prints out the registers including the program counter and an interpretation of the instruction at that location.

The request:

$e

prints out the values of all external variables.

A map exists for each file handled by ADB. The map for the *a.out* file is referenced by **?** whereas the map for *core* file is referenced by **/**. Furthermore, a good rule of thumb is to use **?** for instructions and **/** for data when looking at programs. To print out information about the maps type:

$m

This produces a report of the contents of the maps. More about these maps later.

In our example, it is useful to see the contents of the string pointed to by *charp.* This is done by:

***charp/s**

which says use *charp* as a pointer in the *core* file and print the information as a character string. This printout clearly shows that the character buffer was incorrectly overwritten and helps identify the error. Printing the locations around *charp* shows that the buffer is unchanged but that the pointer is destroyed. Using ADB similarly, we could print information about the arguments to a function. The request:

main.argc/d

prints the decimal *core* image value of the argument *argc* in the function *main.*
The request:

***main.argv,3/o**

prints the octal values of the three consecutive cells pointed to by *argv* in the function *main.* Note that these values are the addresses of the arguments to main. Therefore:

0177770/s

prints the ASCII value of the first argument. Another way to print this value would have been

***"/s**

The " means ditto which remembers the last address typed, in this case *main.argc* ; the ***** instructs ADB to use the address field of the *core* file as a pointer.

The request:

.=o

prints the current address (not its contents) in octal which has been set to the address of the first argument. The current address, dot, is used by ADB to "remember" its current location. It allows the user to reference locations relative to the current address, for example:

.–10/d

3.2. Multiple Functions

Consider the C program illustrated in Figure 3. This program calls functions *f, g,* and *h* until the stack is exhausted and a core image is produced.

Again you can enter the debugger via:

adb

which assumes the names *a.out* and *core* for the executable file and core image file respectively. The request:

$c

will fill a page of backtrace references to *f, g,* and *h.* Figure 4 shows an abbreviated list (typing *DEL* will terminate the output and bring you back to ADB request level).

The request:

,5$C

prints the five most recent activations.

Notice that each function (*f,g,h*) has a counter of the number of times it was called.

The request:

fcnt/d

prints the decimal value of the counter for the function *f.* Similarly *gcnt* and *hcnt* could be printed. To print the value of an automatic variable, for example the decimal value of *x* in the last call of the

function *h,* type:

h.x/d

It is currently not possible in the exported version to print stack frames other than the most recent activation of a function. Therefore, a user can print everything with **$C** or the occurrence of a variable in the most recent call of a function. It is possible with the **$C** request, however, to print the stack frame starting at some address as **address$C.**

3.3. Setting Breakpoints

Consider the C program in Figure 5. This program, which changes tabs into blanks, is adapted from *Software Tools* by Kernighan and Plauger, pp. 18-27.

We will run this program under the control of ADB (see Figure 6a) by:

adb a.out –

Breakpoints are set in the program as:

address:b [request]

The requests:

settab+4:b
fopen+4:b
getc+4:b
tabpos+4:b

set breakpoints at the start of these functions. C does not generate statement labels. Therefore it is currently not possible to plant breakpoints at locations other than function entry points without a knowledge of the code generated by the C compiler. The above addresses are entered as **symbol+4** so that they will appear in any C backtrace since the first instruction of each function is a call to the C save routine (*csv*). Note that some of the functions are from the C library.

To print the location of breakpoints one types:

$b

The display indicates a *count* field. A breakpoint is bypassed *count –1* times before causing a stop. The *command* field indicates the ADB requests to be executed each time the breakpoint is encountered. In our example no *command* fields are present.

By displaying the original instructions at the function *settab* we see that the breakpoint is set after the jsr to the C save routine. We can display the instructions using the ADB request:

settab,5?ia

This request displays five instructions starting at *settab* with the addresses of each location displayed. Another variation is:

settab,5?i

which displays the instructions with only the starting address.

Notice that we accessed the addresses from the *a.out* file with the **?** command. In general when asking for a printout of multiple items, ADB will advance the current address the number of bytes necessary to satisfy the request; in the above example five instructions were displayed and the current address was advanced 18 (decimal) bytes.

To run the program one simply types:

:r

To delete a breakpoint, for instance the entry to the function *settab,* one types:

settab+4:d

To continue execution of the program from the breakpoint type:

:c

Once the program has stopped (in this case at the breakpoint for *fopen),* ADB requests can be used to display the contents of memory. For example:

$C

to display a stack trace, or:

tabs,3/8o

to print three lines of 8 locations each from the array called *tabs.* By this time (at location *fopen)* in the C program, *settab* has been called and should have set a one in every eighth location of *tabs.*

3.4. Advanced Breakpoint Usage

We continue execution of the program with:

:c

See Figure 6b. *Getc* is called three times and the contents of the variable *c* in the function *main* are displayed each time. The single character on the left hand edge is the output from the C program. On the third occurrence of *getc* the program stops. We can look at the full buffer of characters by typing:

ibuf+6/20c

When we continue the program with:

:c

we hit our first breakpoint at *tabpos* since there is a tab following the "This" word of the data.

Several breakpoints of *tabpos* will occur until the program has changed the tab into equivalent blanks. Since we feel that *tabpos* is working, we can remove the breakpoint at that location by:

tabpos+4:d

If the program is continued with:

:c

it resumes normal execution after ADB prints the message

a.out:running

The UNIX quit and interrupt signals act on ADB itself rather than on the program being debugged. If such a signal occurs then the program being debugged is stopped and control is returned to ADB. The signal is saved by ADB and is passed on to the test program if:

:c

is typed. This can be useful when testing interrupt handling routines. The signal is not passed on to the test program if:

:c 0

is typed.

Now let us reset the breakpoint at *settab* and display the instructions located there when we reach the breakpoint. This is accomplished by:

settab+4:b settab,5?ia *

It is also possible to execute the ADB requests for each occurrence of the breakpoint but only stop after

---

* Owing to a bug in early versions of ADB (including the version distributed in Generic 3 UNIX) these statements must be written as:

the third occurrence by typing:

```
getc+4,3:b  main.c?C  *
```

This request will print the local variable *c* in the function *main* at each occurrence of the breakpoint. The semicolon is used to separate multiple ADB requests on a single line.

Warning: setting a breakpoint causes the value of dot to be changed; executing the program under ADB does not change dot. Therefore:

```
settab+4:b  .,5?ia
fopen+4:b
```

will print the last thing dot was set to (in the example *fopen+4*) *not* the current location (*settab+4*) at which the program is executing.

A breakpoint can be overwritten without first deleting the old breakpoint. For example:

```
settab+4:b  settab,5?ia; ptab/o  *
```

could be entered after typing the above requests.

Now the display of breakpoints:

```
$b
```

shows the above request for the *settab* breakpoint. When the breakpoint at *settab* is encountered the ADB requests are executed. Note that the location at *settab+4* has been changed to plant the breakpoint; all the other locations match their original value.

Using the functions, *f, g* and *h* shown in Figure 3, we can follow the execution of each function by planting non-stopping breakpoints. We call ADB with the executable program of Figure 3 as follows:

```
adb ex3 –
```

Suppose we enter the following breakpoints:

```
h+4:b       hcnt/d;  h.hi/;  h.hr/
g+4:b       gcnt/d;  g.gi/;  g.gr/
f+4:b       fcnt/d;  f.fi/;  f.fr/
:r
```

Each request line indicates that the variables are printed in decimal (by the specification **d**). Since the format is not changed, the **d** can be left off all but the first request.

The output in Figure 7 illustrates two points. First, the ADB requests in the breakpoint line are not examined until the program under test is run. That means any errors in those ADB requests is not detected until run time. At the location of the error ADB stops running the program.

The second point is the way ADB handles register variables. ADB uses the symbol table to address variables. Register variables, like *f.fr* above, have pointers to uninitialized places on the stack. Therefore the message "symbol not found".

Another way of getting at the data in this example is to print the variables used in the call as:

```
f+4:b       fcnt/d;  f.a/;  f.b/;  f.fi/
g+4:b       gcnt/d;  g.p/;  g.q/;  g.gi/
:c
```

The operator / was used instead of ? to read values from the *core* file. The output for each function, as

---

```
settab+4:b          settab,5?ia;0
getc+4,3:bmain.c?C;0
settab+4:b          settab,5?ia; ptab/o;0
```

Note that **;0** will set dot to zero and stop at the breakpoint.

shown in Figure 7, has the same format. For the function *f*, for example, it shows the name and value of the *external* variable *fcnt.* It also shows the address on the stack and value of the variables *a, b* and *fi.*

Notice that the addresses on the stack will continue to decrease until no address space is left for program execution at which time (after many pages of output) the program under test aborts. A display with names would be produced by requests like the following:

f+4:b fcnt/d; f.a/"a="d; f.b/"b="d; f.fi/"fi="d

In this format the quoted string is printed literally and the **d** produces a decimal display of the variables. The results are shown in Figure 7.

3.5. Other Breakpoint Facilities

- Arguments and change of standard input and output are passed to a program as:

  :r arg1 arg2 ... <infile >outfile

  This request kills any existing program under test and starts the *a.out* afresh.

- The program being debugged can be single stepped by:

  :s

  If necessary, this request will start up the program being debugged and stop after executing the first instruction.

- ADB allows a program to be entered at a specific address by typing:

  address:r

- The count field can be used to skip the first *n* breakpoints as:

  ,n:r

  The request:

  ,n:c

  may also be used for skipping the first *n* breakpoints when continuing a program.

- A program can be continued at an address different from the breakpoint by:

  address:c

- The program being debugged runs as a separate process and can be killed by:

  :k

4. Maps

UNIX supports several executable file formats. These are used to tell the loader how to load the program file. File type 407 is the most common and is generated by a C compiler invocation such as **cc pgm.c**. A 410 file is produced by a C compiler command of the form **cc -n pgm.c**, whereas a 411 file is produced by **cc -i pgm.c**. ADB interprets these different file formats and provides access to the different segments through a set of maps (see Figure 8). To print the maps type:

$m

In 407 files, both text (instructions) and data are intermixed. This makes it impossible for ADB to differentiate data from instructions and some of the printed symbolic addresses look incorrect; for example, printing data addresses as offsets from routines.

In 410 files (shared text), the instructions are separated from data and **?*** accesses the data part of the *a.out* file. The **?*** request tells ADB to use the second part of the map in the *a.out* file. Accessing data in the *core* file shows the data after it was modified by the execution of the program. Notice also that the data segment may have grown during program execution.

In 411 files (separated I & D space), the instructions and data are also separated. However, in this case, since data is mapped through a separate set of segmentation registers, the base of the data segment is also relative to address zero. In this case since the addresses overlap it is necessary to use the **?*** operator to access the data space of the *a.out* file. In both 410 and 411 files the corresponding core file does not contain the program text.

Figure 9 shows the display of three maps for the same program linked as a 407, 410, 411 respectively. The b, e, and f fields are used by ADB to map addresses into file addresses. The "f1" field is the length of the header at the beginning of the file (020 bytes for an *a.out* file and 02000 bytes for a *core* file). The "f2" field is the displacement from the beginning of the file to the data. For a 407 file with mixed text and data this is the same as the length of the header; for 410 and 411 files this is the length of the header plus the size of the text portion.

The "b" and "e" fields are the starting and ending locations for a segment. Given an address, A, the location in the file (either *a.out* or *core*) is calculated as:

b1≤A≤e1 ⇒ file address = (A–b1)+f1
b2≤A≤e2 ⇒ file address = (A–b2)+f2

A user can access locations by using the ADB defined variables. The **$v** request prints the variables initialized by ADB:

b	**base address of data segment**
d	**length of the data segment**
s	**length of the stack**
t	**length of the text**
m	**execution type (407,410,411)**

In Figure 9 those variables not present are zero. Use can be made of these variables by expressions such as:

```
<b
```

in the address field. Similarly the value of the variable can be changed by an assignment request such as:

```
02000>b
```

that sets **b** to octal 2000. These variables are useful to know if the file under examination is an executable or *core* image file.

ADB reads the header of the *core* image file to find the values for these variables. If the second file specified does not seem to be a *core* file, or if it is missing then the header of the executable file is used instead.

5. Advanced Usage

It is possible with ADB to combine formatting requests to provide elaborate displays. Below are several examples.

5.1. Formatted dump

The line:

```
<b,–1/4o4^8Cn
```

prints 4 octal words followed by their ASCII interpretation from the data space of the core image file. Broken down, the various request pieces mean:

<b	The base address of the data segment.
<b,–1	Print from the base address to the end of file. A negative count is used here and elsewhere to loop indefinitely or until some error condition (like end of file) is detected.

The format **4o4^8Cn** is broken down as follows:

4o	Print 4 octal locations.
4^	Backup the current address 4 locations (to the original start of the field).
8C	Print 8 consecutive characters using an escape convention; each character in the range 0 to 037 is printed as @ followed by the corresponding character in the range 0140 to 0177. An @ is printed as @@.
n	Print a newline.

The request:

<b,<d/4o4^8Cn

could have been used instead to allow the printing to stop at the end of the data segment (<d provides the data segment size in bytes).

The formatting requests can be combined with ADB's ability to read in a script to produce a core image dump script. ADB is invoked as:

adb a.out core < dump

to read in a script file, *dump,* of requests. An example of such a script is:

```
120$w
4095$s
$v
=3n
$m
=3n"C Stack Backtrace"
$C
=3n"C External Variables"
$e
=3n"Registers"
$r
0$s
=3n"Data Segment"
<b,–1/8ona
```

The request **120$w** sets the width of the output to 120 characters (normally, the width is 80 characters). ADB attempts to print addresses as:

symbol + offset

The request **4095$s** increases the maximum permissible offset to the nearest symbolic address from 255 (default) to 4095. The request **=** can be used to print literal strings. Thus, headings are provided in this *dump* program with requests of the form:

=3n"C Stack Backtrace"

that spaces three lines and prints the literal string. The request **$v** prints all non-zero ADB variables (see Figure 8). The request **0$s** sets the maximum offset for symbol matches to zero thus suppressing

the printing of symbolic labels in favor of octal values. Note that this is only done for the printing of the data segment. The request:

<b,–1/8ona

prints a dump from the base of the data segment to the end of file with an octal address field and eight octal numbers per line.

Figure 11 shows the results of some formatting requests on the C program of Figure 10.

5.2. Directory Dump

As another illustration (Figure 12) consider a set of requests to dump the contents of a directory (which is made up of an integer *inumber* followed by a 14 character name):

```
adb dir –
=n8t"Inum"8t"Name"
0,–1? u8t14cn
```

In this example, the **u** prints the *inumber* as an unsigned decimal integer, the **8t** means that ADB will space to the next multiple of 8 on the output line, and the **14c** prints the 14 character file name.

5.3. Ilist Dump

Similarly the contents of the *ilist* of a file system, (e.g. /dev/src, on UNIX systems distributed by the UNIX Support Group; see UNIX Programmer's Manual Section V) could be dumped with the following set of requests:

```
adb /dev/src –
02000>b
?m <b
<b,–1?"flags"8ton"links,uid,gid"8t3bn",size"8tbrdn"addr"8t8un"times"8t2Y2na
```

In this example the value of the base for the map was changed to 02000 (by saying **?m<b**) since that is the start of an *ilist* within a file system. An artifice (**brd** above) was used to print the 24 bit size field as a byte, a space, and a decimal integer. The last access time and last modify time are printed with the **2Y** operator. Figure 12 shows portions of these requests as applied to a directory and file system.

5.4. Converting values

ADB may be used to convert values from one representation to another. For example:

072 = odx

will print

```
072             58              #3a
```

which is the octal, decimal and hexadecimal representations of 072 (octal). The format is remembered so that typing subsequent numbers will print them in the given formats. Character values may be converted similarly, for example:

'a' = co

prints

```
a               0141
```

It may also be used to evaluate expressions but be warned that all binary operators have the same precedence which is lower than that for unary operators.

6. Patching

Patching files with ADB is accomplished with the *write,* **w** or **W**, request (which is not like the *ed* editor write command). This is often used in conjunction with the *locate,* **l** or **L** request. In general, the request syntax for **l** and **w** are similar as follows:

?l value

The request **l** is used to match on two bytes, **L** is used for four bytes. The request **w** is used to write two bytes, whereas **W** writes four bytes. The **value** field in either *locate* or *write* requests is an expression. Therefore, decimal and octal numbers, or character strings are supported.

In order to modify a file, ADB must be called as:

adb –w file1 file2

When called with this option, *file1* and *file2* are created if necessary and opened for both reading and writing.

For example, consider the C program shown in Figure 10. We can change the word "This" to "The " in the executable file for this program, *ex7*, by using the following requests:

```
adb –w ex7 –
?l 'Th'
?W 'The '
```

The request **?l** starts at dot and stops at the first match of "Th" having set dot to the address of the location found. Note the use of **?** to write to the *a.out* file. The form **?*** would have been used for a 411 file.

More frequently the request will be typed as:

?l 'Th'; ?s

and locates the first occurrence of "Th" and print the entire string. Execution of this ADB request will set dot to the address of the "Th" characters.

As another example of the utility of the patching facility, consider a C program that has an internal logic flag. The flag could be set by the user through ADB and the program run. For example:

```
adb a.out –
:s arg1 arg2
flag/w 1
:c
```

The **:s** request is normally used to single step through a process or start a process in single step mode. In this case it starts *a.out* as a subprocess with arguments **arg1** and **arg2**. If there is a subprocess running ADB writes to it rather than to the file so the **w** request causes *flag* to be changed in the memory of the subprocess.

7. Anomalies

Below is a list of some strange things that users should be aware of.

1. Function calls and arguments are put on the stack by the C save routine. Putting breakpoints at the entry point to routines means that the function appears not to have been called when the breakpoint occurs.
2. When printing addresses, ADB uses either text or data symbols from the *a.out* file. This sometimes causes unexpected symbol names to be printed with data (e.g. *savr5+022*). This does not happen if **?** is used for text (instructions) and **/** for data.
3. ADB cannot handle C register variables in the most recently activated function.

8. Acknowledgements

The authors are grateful for the thoughtful comments on how to organize this document from R. B. Brandt, E. N. Pinson and B. A. Tague. D. M. Ritchie made the system changes necessary to accommodate tracing within ADB. He also participated in discussions during the writing of ADB. His earlier work with DB and CDB led to many of the features found in ADB.

9. References

1. D. M. Ritchie and K. Thompson, ‘‘The UNIX Time-Sharing System,’’ CACM, July, 1974.
2. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, 1978.
3. K. Thompson and D. M. Ritchie, UNIX Programmer’s Manual - 7th Edition, 1978.
4. B. W. Kernighan and P. J. Plauger, *Software Tools,* Addison-Wesley, 1976.

Figure 1: C program with pointer bug

```
struct buf {
        int fildes;
        int nleft;
        char *nextp;
        char buff[512];
        }bb;
struct buf *obuf;

char *charp "this is a sentence.";

main(argc,argv)
int argc;
char **argv;
{
        char    cc;

        if(argc < 2) {
                printf("Input file missing\n");
                exit(8);
        }

        if((fcreat(argv[1],obuf)) < 0){
                printf("%s : not found\n", argv[1]);
                exit(8);
        }
        charp = ´T´;
printf("debug 1 %s\n",charp);
        while(cc=  *charp++)
                putc(cc,obuf);
        fflush(obuf);
}
```

Figure 2: ADB output for C program of Figure 1

```
adb a.out core
$c
~main(02,0177762)
$C
~main(02,0177762)
            argc:       02
            argv:       0177762
            cc:         02124
$r
ps      0170010
pc      0204    ~main+0152
sp      0177740
r5      0177752
r4      01
r3      0
r2      0
r1      0
r0      0124
~main+0152:     mov     _obuf,(sp)
$e
savr5:      0
_obuf:      0
_charp:     0124
_errno:     0
_fout:      0
$m
text map    `ex1´
b1 = 0              e1    = 02360           f1 = 020
b2 = 0              e2    = 02360           f2 = 020
data map    `core1´
b1 = 0              e1    = 03500           f1 = 02000
b2 = 0175400        e2    = 0200000               f2 = 05500
*charp/s
0124:           TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTLx     Nh@x&_
~
charp/s
_charp:         T

_charp+02:      this is a sentence.

_charp+026:     Input file missing
main.argc/d
0177756: 2
*main.argv/3o
0177762: 0177770  0177776  0177777
0177770/s
0177770: a.out
*main.argv/3o
0177762: 0177770  0177776  0177777
*''/s
0177770: a.out
 .=o
                0177770
 .–10/d
0177756: 2
$q
```

Figure 3: Multiple function C program for stack trace illustration

```
int     fcnt,gcnt,hcnt;
h(x,y)
{
        int hi; register int hr;
        hi = x+1;
        hr = x−y+1;
        hcnt++ ;
        hj:
        f(hr,hi);
}

g(p,q)
{
        int gi; register int gr;
        gi = q−p;
        gr = q−p+1;
        gcnt++ ;
        gj:
        h(gr,gi);
}

f(a,b)
{
        int fi; register int fr;
        fi = a+2*b;
        fr = a+b;
        fcnt++ ;
        fj:
        g(fr,fi);
}

main()
{
        f(1,1);
}
```

Figure 4: ADB output for C program of Figure 3

```
adb
$c
~h(04452,04451)
~g(04453,011124)
~f(02,04451)
~h(04450,04447)
~g(04451,011120)
~f(02,04447)
~h(04446,04445)
~g(04447,011114)
~f(02,04445)
~h(04444,04443)
HIT DEL KEY
adb
,5$C
~h(04452,04451)
        x:      04452
        y:      04451
        hi:     ?
~g(04453,011124)
        p:      04453
        q:      011124
        gi:     04451
        gr:     ?
~f(02,04451)
        a:      02
        b:      04451
        fi:     011124
        fr:     04453
~h(04450,04447)
        x:      04450
        y:      04447
        hi:     04451
        hr:     02
~g(04451,011120)
        p:      04451
        q:      011120
        gi:     04447
        gr:     04450
fcnt/d
_fcnt:          1173
gcnt/d
_gcnt:          1173
hcnt/d
_hcnt:          1172
h.x/d
022004:         2346
$q
```

Figure 5: C program to decode tabs

```
#define MAXLINE 80
#define YES             1
#define NO              0
#define TABSP           8

char    input[] "data";
char    ibuf[518];
int     tabs[MAXLINE];

main()
{
        int col, *ptab;
        char c;

        ptab = tabs;
        settab(ptab);       /*Set initial tab stops */
        col = 1;
        if(fopen(input,ibuf) < 0) {
                printf("%s : not found\n",input);
                exit(8);
        }
        while((c = getc(ibuf)) != -1) {
                switch(c) {
                        case '\t': /* TAB */
                                while(tabpos(col) != YES) {
                                        putchar(' ');       /* put BLANK */
                                        col++ ;
                                }
                                break;
                        case '\n': /*NEWLINE */
                                putchar('\n');
                                col = 1;
                                break;
                        default:
                                putchar(c);
                                col++ ;
                }
        }
}

/* Tabpos return YES if col is a tab stop */
tabpos(col)
int col;
{
        if(col > MAXLINE)
                return(YES);
        else
                return(tabs[col]);
}

/* Settab - Set initial tab stops */
settab(tabp)
int *tabp;
{
        int i;

        for(i = 0; i<= MAXLINE; i++)
                (i%TABSP) ? (tabs[i] = NO) : (tabs[i] = YES);
}
```

Figure 6a: ADB output for C program of Figure 5

```
adb a.out –
settab+4:b
fopen+4:b
getc+4:b
tabpos+4:b
$b
breakpoints
count    bkpt              command
1        ~tabpos+04
1        _getc+04
1        _fopen+04
1        ~settab+04
settab,5?ia
~settab:        jsr     r5,csv
~settab+04:     tst     −(sp)
~settab+06:     clr     0177770(r5)
~settab+012:    cmp     $0120,0177770(r5)
~settab+020:    blt     ~settab+076
~settab+022:
settab,5?i
~settab:        jsr     r5,csv
                tst     −(sp)
                clr     0177770(r5)
                cmp     $0120,0177770(r5)
                blt     ~settab+076
:r
a.out: running
breakpoint      ~settab+04:     tst     −(sp)
settab+4:d
:c
a.out: running
breakpoint      _fopen+04:      mov     04(r5),nulstr+012
$C
_fopen(02302,02472)
~main(01,0177770)
        col:    01
        c:      0
        ptab:   03500
tabs,3/8o
03500:          01      0       0       0       0       0       0       0
                01      0       0       0       0       0       0       0
                01      0       0       0       0       0       0       0
```

Figure 6b: ADB output for C program of Figure 5

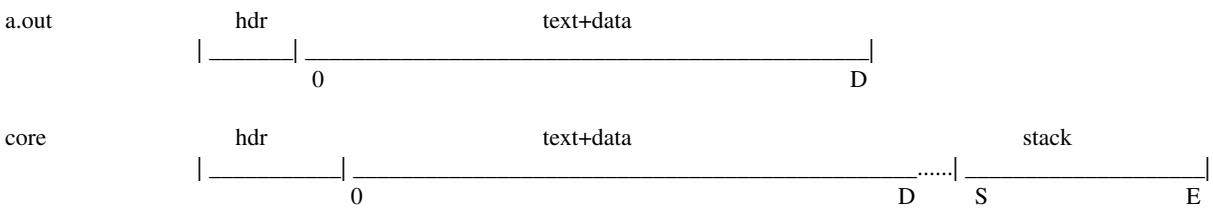
```
:c
a.out: running
breakpoint          _getc+04:          mov     04(r5),r1
ibuf+6/20c
__cleanu+0202:              This    is      a test  of
:c
a.out: running
breakpoint          ~tabpos+04:        cmp     $0120,04(r5)
tabpos+4:d
settab+4:b  settab,5?ia
settab+4:b  settab,5?ia;  0
getc+4,3:b  main.c?C;  0
settab+4:b  settab,5?ia;  ptab/o;  0
$b
breakpoints
count   bkpt                command
1       ~tabpos+04
3       _getc+04 main.c?C;0
1       _fopen+04
1       ~settab+04          settab,5?ia;ptab?o;0
~settab:            jsr     r5,csv
~settab+04:         bpt
~settab+06:         clr     0177770(r5)
~settab+012:        cmp     $0120,0177770(r5)
~settab+020:        blt     ~settab+076
~settab+022:
0177766: 0177770
0177744: @`
T0177744:           T
h0177744:           h
i0177744:           i
s0177744:           s
```

Figure 7: ADB output for C program with breakpoints

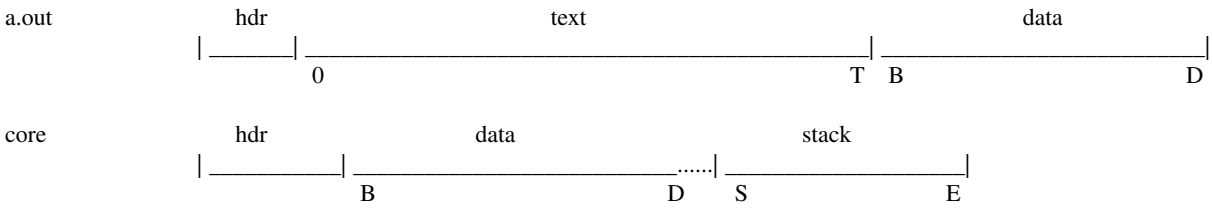
```
adb ex3 –
h+4:b hcnt/d; h.hi/; h.hr/
g+4:b gcnt/d; g.gi/; g.gr/
f+4:b fcnt/d; f.fi/; f.fr/
:r
ex3: running
_fcnt:          0
0177732:        214
symbol not found
f+4:b fcnt/d; f.a/; f.b/; f.fi/
g+4:b gcnt/d; g.p/; g.q/; g.gi/
h+4:b hcnt/d; h.x/; h.y/; h.hi/
:c
ex3: running
_fcnt:          0
0177746:        1
0177750:        1
0177732:        214
_gcnt:          0
0177726:        2
0177730:        3
0177712:        214
_hcnt:          0
0177706:        2
0177710:        1
0177672:        214
_fcnt:          1
0177666:        2
0177670:        3
0177652:        214
_gcnt:          1
0177646:        5
0177650:        8
0177632:        214
HIT DEL
f+4:b fcnt/d; f.a/"a = "d; f.b/"b = "d; f.fi/"fi = "d
g+4:b gcnt/d; g.p/"p = "d; g.q/"q = "d; g.gi/"gi = "d
h+4:b hcnt/d; h.x/"x = "d; h.y/"h = "d; h.hi/"hi = "d
:r
ex3: running
_fcnt:          0
0177746:        a = 1
0177750:        b = 1
0177732:        fi = 214
_gcnt:          0
0177726:        p = 2
0177730:        q = 3
0177712:        gi = 214
_hcnt:          0
0177706:        x = 2
0177710:        y = 1
0177672:        hi = 214
_fcnt:          1
0177666:        a = 2
0177670:        b = 3
0177652:        fi = 214
HIT DEL
$q
```

Figure 8: ADB address maps

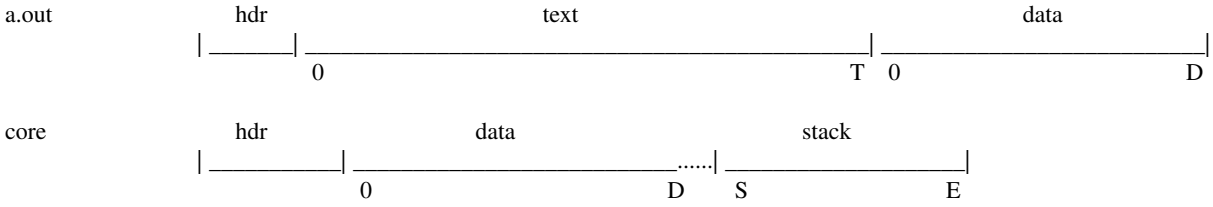
407 files



410 files (shared text)



411 files (separated I and D space)



The following *adb* variables are set.

		407	410	411
b	base of data	0	B	0
d	length of data	D	D–B	D
s	length of stack	S	S	S
t	length of text	0	T	T

Figure 9: ADB output for maps

```
adb map407 core407
$m
text map    `map407´
b1 = 0              e1      = 0256          f1 = 020
b2 = 0              e2      = 0256          f2 = 020
data map    `core407´
b1 = 0              e1      = 0300          f1 = 02000
b2 = 0175400        e2      = 0200000   f2 = 02300
$v
variables
d = 0300
m = 0407
s = 02400
$q

adb map410 core410
$m
text map    `map410´
b1 = 0              e1      = 0200          f1 = 020
b2 = 020000         e2      = 020116    f2 = 0220
data map    `core410´
b1 = 020000         e1      = 020200    f1 = 02000
b2 = 0175400        e2      = 0200000   f2 = 02200
$v
variables
b = 020000
d = 0200
m = 0410
s = 02400
t = 0200
$q

adb map411 core411
$m
text map    `map411´
b1 = 0              e1      = 0200          f1 = 020
b2 = 0              e2      = 0116          f2 = 0220
data map    `core411´
b1 = 0              e1      = 0200          f1 = 02000
b2 = 0175400        e2      = 0200000   f2 = 02200
$v
variables
d = 0200
m = 0411
s = 02400
t = 0200
$q
```

Figure 10: Simple C program for illustrating formatting and patching

```
char    str1[]    "This is a character string";
int     one       1;
int     number    456;
long    lnum      1234;
float   fpt       1.25;
char    str2[]    "This is the second character string";
main()
{
        one = 2;
}
```

Figure 11: ADB output illustrating fancy formats

```
adb map410 core410
<b,–1/8ona
020000:          0     064124     071551     064440     020163     020141     064143071141

_str1+016:  061541     062564     020162     072163     064562     063556     0     02

_number:
_number:    0710 0     02322      040240     0     064124     071551     064440

_str2+06:   020163     064164     020145     062563     067543     062156     061440060550

_str2+026:  060562     072143     071145     071440     071164     067151     0147 0

savr5+02:   0     0     0     0     0     0     0     0

<b,20/4o4^8Cn
020000:          0     064124     071551     064440     @`@`This i
            020163     020141     064143     071141     s a char
            061541     062564     020162     072163     acter st
            064562     063556     0     02   ring@`@`@b@`

_number:    0710 0     02322      040240     H@a@`@`R@d @@
            0     064124     071551     064440     @`@`This i
            020163     064164     020145     062563     s the se
            067543     062156     061440     060550     cond cha
            060562     072143     071145     071440     racter s
            071164     067151     0147 0     tring@`@`@`
            0     0     0     0     @`@`@`@`@`@`@`@`
            0     0     0     0     @`@`@`@`@`@`@`@`
data address not found
<b,20/4o4^8t8cna
020000:          0     064124     071551     064440          This i
_str1+06:   020163     020141     064143     071141          s a char
_str1+016:  061541     062564     020162     072163          acter st
_str1+026:  064562     063556     0     02          ring
_number:
_number:    0710 0     02322      040240          HR
_fpt+02:    0     064124     071551     064440          This i
_str2+06:   020163     064164     020145     062563          s the se
_str2+016:  067543     062156     061440     060550          cond cha
_str2+026:  060562     072143     071145     071440          racter s
_str2+036:  071164     067151     0147 0          tring
savr5+02:   0     0     0     0
savr5+012:  0     0     0     0
data address not found
<b,10/2b8t^2cn
020000:          0     0

_str1:      0124 0150      Th
            0151 0163      is
            040  0151       i
            0163 040       s
            0141 040       a
            0143 0150      ch
            0141 0162      ar
            0141 0143      ac
            0164 0145      te
$Q
```

Figure 12: Directory and inode dumps

```
adb dir –
=nt"Inode"t"Name"
0,–1?ut14cn

            Inode Name
0:          652   .
            82    ..
            5971  cap.c
            5323  cap
            0     pp



adb /dev/src –
02000>b
?m<b
new map     `/dev/src´
b1 = 02000    e1     = 0100000000     f1 = 0
b2 = 0              e2     = 0          f2 = 0
$v
variables
b = 02000
<b,–1?"flags"8ton"links,uid,gid"8t3bn"size"8tbrdn"addr"8t8un"times"8t2Y2na
02000:          flags  073145
            links,uid,gid      0163  0164  0141
            size   0162  10356
            addr   28770       8236  25956       27766       25455       8236  25956       25206
            times 1976 Feb 5 08:34:56    1975 Dec 28 10:55:15

02040:          flags  024555
            links,uid,gid      012   0163  0164
            size   0162  25461
            addr   8308  30050       8294  25130       15216       26890       29806       10784
            times 1976 Aug 17 12:16:51   1976 Aug 17 12:16:51

02100:          flags  05173
            links,uid,gid      011   0162  0145
            size   0147  29545
            addr   25972       8306  28265       8308  25642       15216       2314  25970
            times 1977 Apr 2 08:58:01    1977 Feb 5 10:21:44
```

ADB Summary

Command Summary

a) formatted printing

? *format*	print from *a.out* file according to *format*
/ *format*	print from *core* file according to *format*
= *format*	print the value of *dot*
?w expr	write expression into *a.out* file
/w expr	write expression into *core* file
?l expr	locate expression in *a.out* file

b) breakpoint and program control

:b	set breakpoint at *dot*
:c	continue running program
:d	delete breakpoint
:k	kill the program being debugged
:r	run *a.out* file under ADB control
:s	single step

c) miscellaneous printing

$b	print current breakpoints
$c	C stack trace
$e	external variables
$f	floating registers
$m	print ADB segment maps
$q	exit from ADB
$r	general registers
$s	set offset for symbol match
$v	print ADB variables
$w	set output line width

d) calling the shell

!	call *shell* to read rest of line

e) assignment to variables

>*name*	assign dot to variable or register *name*

Format Summary

a	the value of dot
b	one byte in octal
c	one byte as a character
d	one word in decimal
f	two words in floating point
i	PDP 11 instruction
o	one word in octal
n	print a newline
r	print a blank space
s	a null terminated character string
***n*t**	move to next *n* space tab
u	one word as unsigned integer
x	hexadecimal
Y	date
^	backup dot
"..."	print string

Expression Summary

a) expression components

decimal integer	e.g. 256
octal integer	e.g. 0277
hexadecimal	e.g. #ff
symbols	e.g. flag _main main.argc
variables	e.g. <b
registers	e.g. <pc <r0
(expression)	expression grouping

b) dyadic operators

+	add
–	subtract
*	multiply
%	integer division
&	bitwise and
|	bitwise or
#	round up to the next multiple

c) monadic operators

~	not
*	contents of location
–	integer negate

PMake — A Tutorial

Adam de Boor

Berkeley Softworks
2150 Shattuck Ave, Penthouse
Berkeley, CA 94704
adam@bsw.uu.net
...!uunet!bsw!adam

1. Introduction

PMake is a program for creating other programs, or anything else you can think of for it to do. The basic idea behind PMake is that, for any given system, be it a program or a document or whatever, there will be some files that depend on the state of other files (on when they were last modified). PMake takes these dependencies, which you must specify, and uses them to build whatever it is you want it to build.

PMake is almost fully-compatible with Make, with which you may already be familiar. PMake's most important feature is its ability to run several different jobs at once, making the creation of systems considerably faster. It also has a great deal more functionality than Make. Throughout the text, whenever something is mentioned that is an important difference between PMake and Make (i.e. something that will cause a makefile to fail if you don't do something about it), or is simply important, it will be flagged with a little sign in the left margin, like this:

NOTE

This tutorial is divided into three main sections corresponding to basic, intermediate and advanced PMake usage. If you already know Make well, you will only need to skim chapter 2 (there are some aspects of PMake that I consider basic to its use that didn't exist in Make). Things in chapter 3 make life much easier, while those in chapter 4 are strictly for those who know what they are doing. Chapter 5 has definitions for the jargon I use and chapter 6 contains possible solutions to the problems presented throughout the tutorial.

2. The Basics of PMake

PMake takes as input a file that tells a) which files depend on which other files to be complete and b) what to do about files that are ''out-of-date.'' This file is known as a ''makefile'' and is usually kept in the top-most directory of the system to be built. While you can call the makefile anything you want, PMake will look for `Makefile` and `makefile` (in that order) in the current directory if you don't tell it otherwise. To specify a different makefile, use the **–f** flag (e.g. ''`pmake -f program.mk`'').

A makefile has four different types of lines in it:

- File dependency specifications
- Creation commands
- Variable assignments
- Comments, include statements and conditional directives

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies. The University of California, Berkeley Softworks, and Adam de Boor make no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

Any line may be continued over multiple lines by ending it with a backslash. The backslash, following newline and any initial whitespace on the following line are compressed into a single space before the input line is examined by PMake.

2.1. Dependency Lines

As mentioned in the introduction, in any system, there are dependencies between the files that make up the system. For instance, in a program made up of several C source files and one header file, the C files will need to be re-compiled should the header file be changed. For a document of several chapters and one macro file, the chapters will need to be reprocessed if any of the macros changes. These are dependencies and are specified by means of dependency lines in the makefile.

On a dependency line, there are targets and sources, separated by a one- or two-character operator. The targets "depend" on the sources and are usually created from them. Any number of targets and sources may be specified on a dependency line. All the targets in the line are made to depend on all the sources. Targets and sources need not be actual files, but every source must be either an actual file or another target in the makefile. If you run out of room, use a backslash at the end of the line to continue onto the next one.

Any file may be a target and any file may be a source, but the relationship between the two (or however many) is determined by the "operator" that separates them. Three types of operators exist: one specifies that the datedness of a target is determined by the state of its sources, while another specifies other files (the sources) that need to be dealt with before the target can be re-created. The third operator is very similar to the first, with the additional condition that the target is out-of-date if it has no sources. These operations are represented by the colon, the exclamation point and the double-colon, respectively, and are mutually exclusive. Their exact semantics are as follows:

: If a colon is used, a target on the line is considered to be "out-of-date" (and in need of creation) if

- any of the sources has been modified more recently than the target, or
- the target doesn't exist.

Under this operation, steps will be taken to re-create the target only if it is found to be out-of-date by using these two rules.

! If an exclamation point is used, the target will always be re-created, but this will not happen until all of its sources have been examined and re-created, if necessary.

:: If a double-colon is used, a target is out-of-date if:

- any of the sources has been modified more recently than the target, or
- the target doesn't exist, or
- the target has no sources.

If the target is out-of-date according to these rules, it will be re-created. This operator also does something else to the targets, but I'll go into that in the next section ("Shell Commands").

Enough words, now for an example. Take that C program I mentioned earlier. Say there are three C files (`a.c`, `b.c` and `c.c`) each of which includes the file `defs.h`. The dependencies between the files could then be expressed as follows:

```
program          : a.o b.o c.o
a.o b.o c.o      : defs.h
a.o              : a.c
b.o              : b.c
c.o              : c.c
```

You may be wondering at this point, where `a.o`, `b.o` and `c.o` came in and why *they* depend on `defs.h` and the C files don't. The reason is quite simple: `program` cannot be made by linking together .c files — it must be made from .o files. Likewise, if you change `defs.h`, it isn't the .c files that need to be re-created, it's the .o files. If you think of dependencies in these terms — which files

(targets) need to be created from which files (sources) — you should have no problems.

An important thing to notice about the above example, is that all the .o files appear as targets on more than one line. This is perfectly all right: the target is made to depend on all the sources mentioned on all the dependency lines. E.g. `a.o` depends on both `defs.h` and `a.c`.

NOTE

The order of the dependency lines in the makefile is important: the first target on the first dependency line in the makefile will be the one that gets made if you don't say otherwise. That's why `program` comes first in the example makefile, above.

Both targets and sources may contain the standard C-Shell wildcard characters (`{`, `}`, `*`, `?`, `[`, and `]`), but the non-curly-brace ones may only appear in the final component (the file portion) of the target or source. The characters mean the following things:

{} These enclose a comma-separated list of options and cause the pattern to be expanded once for each element of the list. Each expansion contains a different element. For example, `src/{whiffle,beep,fish}.c` expands to the three words `src/whiffle.c`, `src/beep.c`, and `src/fish.c`. These braces may be nested and, unlike the other wildcard characters, the resulting words need not be actual files. All other wildcard characters are expanded using the files that exist when PMake is started.

***** This matches zero or more characters of any sort. `src/*.c` will expand to the same three words as above as long as `src` contains those three files (and no other files that end in `.c`).

? Matches any single character.

[] This is known as a character class and contains either a list of single characters, or a series of character ranges (`a-z`, for example means all characters between a and z), or both. It matches any single character contained in the list. E.g. `[A-Za-z]` will match all letters, while `[0123456789]` will match all numbers.

2.2. Shell Commands

"Isn't that nice," you say to yourself, "but how are files actually 're-created,' as he likes to spell it?" The re-creation is accomplished by commands you place in the makefile. These commands are passed to the Bourne shell (better known as "/bin/sh") to be executed and are expected to do what's necessary to update the target file (PMake doesn't actually check to see if the target was created. It just assumes it's there).

Shell commands in a makefile look a lot like shell commands you would type at a terminal, with one important exception: each command in a makefile *must* be preceded by at least one tab.

Each target has associated with it a shell script made up of one or more of these shell commands. The creation script for a target should immediately follow the dependency line for that target. While any given target may appear on more than one dependency line, only one of these dependency lines may be followed by a creation script, unless the '::' operator was used on the dependency line.

NOTE

If the double-colon was used, each dependency line for the target may be followed by a shell script. That script will only be executed if the target on the associated dependency line is out-of-date with respect to the sources on that line, according to the rules I gave earlier. I'll give you a good example of this later on.

To expand on the earlier makefile, you might add commands as follows:

```
program         : a.o b.o c.o
        cc a.o b.o c.o -o program
a.o b.o c.o     : defs.h
a.o             : a.c
        cc -c a.c
b.o             : b.c
        cc -c b.c
c.o             : c.c
        cc -c c.c
```

Something you should remember when writing a makefile is, the commands will be executed if the *target* on the dependency line is out-of-date, not the sources. In this example, the command "`cc -c a.c`" will be executed if `a.o` is out-of-date. Because of the ':' operator, this means that should `a.c` *or* `defs.h` have been modified more recently than `a.o`, the command will be executed (`a.o` will be considered out-of-date).

Remember how I said the only difference between a makefile shell command and a regular shell command was the leading tab? I lied. There is another way in which makefile commands differ from regular ones. The first two characters after the initial whitespace are treated specially. If they are any combination of '@' and '–', they cause PMake to do different things.

In most cases, shell commands are printed before they're actually executed. This is to keep you informed of what's going on. If an '@' appears, however, this echoing is suppressed. In the case of an `echo` command, say "`echo Linking index`," it would be rather silly to see

```
echo Linking index
Linking index
```

so PMake allows you to place an '@' before the command ("`@echo Linking index`") to prevent the command from being printed.

The other special character is the '–'. In case you didn't know, shell commands finish with a certain "exit status." This status is made available by the operating system to whatever program invoked the command. Normally this status will be 0 if everything went ok and non-zero if something went wrong. For this reason, PMake will consider an error to have occurred if one of the shells it invokes returns a non-zero status. When it detects an error, PMake's usual action is to abort whatever it's doing and exit with a non-zero status itself (any other targets that were being created will continue being made, but nothing new will be started. PMake will exit after the last job finishes). This behavior can be altered, however, by placing a '–' at the front of a command ("`-mv index index.old`"), certain command-line arguments, or doing other things, to be detailed later. In such a case, the non-zero status is simply ignored and PMake keeps chugging along.

NOTE

Because all the commands are given to a single shell to execute, such things as setting shell variables, changing directories, etc., last beyond the command in which they are found. This also allows shell compound commands (like `for` loops) to be entered in a natural manner. Since this could cause problems for some makefiles that depend on each command being executed by a single shell, PMake has a **–B** flag (it stands for backwards-compatible) that forces each command to be given to a separate shell. It also does several other things, all of which I discourage since they are now old-fashioned. . . .

NOTE

A target's shell script is fed to the shell on its (the shell's) input stream. This means that any commands, such as `ci` that need to get input from the terminal won't work right — they'll get the shell's input, something they probably won't find to their liking. A simple way around this is to give a command like this:

```
ci $(SRCS) < /dev/tty
```

This would force the program's input to come from the terminal. If you can't do this for some reason, your only other alternative is to use PMake in its fullest compatibility mode. See **Compatibility** in chapter 4.

2.3. Variables

PMake, like Make before it, has the ability to save text in variables to be recalled later at your convenience. Variables in PMake are used much like variables in the shell and, by tradition, consist of all upper-case letters (you don't *have* to use all upper-case letters. In fact there's nothing to stop you from calling a variable `@^&$%$`. Just tradition). Variables are assigned-to using lines of the form

```
VARIABLE = value
```

appended-to by

```
VARIABLE += value
```

conditionally assigned-to (if the variable isn't already defined) by

```
VARIABLE ?= value
```

and assigned-to with expansion (i.e. the value is expanded (see below) before being assigned to the variable—useful for placing a value at the beginning of a variable, or other things) by

```
VARIABLE := value
```

Any whitespace before *value* is stripped off. When appending, a space is placed between the old value and the stuff being appended.

The final way a variable may be assigned to is using

```
VARIABLE != shell-command
```

In this case, *shell-command* has all its variables expanded (see below) and is passed off to a shell to execute. The output of the shell is then placed in the variable. Any newlines (other than the final one) are replaced by spaces before the assignment is made. This is typically used to find the current directory via a line like:

```
CWD             != pwd
```

Note: this is intended to be used to execute commands that produce small amounts of output (e.g. "pwd"). The implementation is less than intelligent and will likely freeze if you execute something that produces thousands of bytes of output (8 Kb is the limit on many UNIX systems).

The value of a variable may be retrieved by enclosing the variable name in parentheses or curly braces and preceeding the whole thing with a dollar sign.

For example, to set the variable CFLAGS to the string "`-I/sprite/src/lib/libc -O`," you would place a line

```
CFLAGS = -I/sprite/src/lib/libc -O
```

in the makefile and use the word `$(CFLAGS)` wherever you would like the string `-I/sprite/src/lib/libc -O` to appear. This is called variable expansion.

NOTE

Unlike Make, PMake will not expand a variable unless it knows the variable exists. E.g. if you have a `${i}` in a shell command and you have not assigned a value to the variable `i` (the empty string is considered a value, by the way), where Make would have substituted the empty string, PMake will leave the `${i}` alone. To keep PMake from substituting for a variable it knows, precede the dollar sign with another dollar sign. (e.g. to pass `${HOME}` to the shell, use `$${HOME}`). This causes PMake, in effect, to expand the `$` macro, which expands to a single `$`. For compatibility, Make's style of variable expansion will be used if you invoke PMake with any of the compatibility flags (**–V**, **–B** or **–M**. The **–V** flag alters just the variable expansion).

There are two different times at which variable expansion occurs: When parsing a dependency line, the expansion occurs immediately upon reading the line. If any variable used on a dependency line is undefined, PMake will print a message and exit. Variables in shell commands are expanded when the command is executed. Variables used inside another variable are expanded whenever the outer variable is expanded (the expansion of an inner variable has no effect on the outer variable. I.e. if the outer

variable is used on a dependency line and in a shell command, and the inner variable changes value between when the dependency line is read and the shell command is executed, two different values will be substituted for the outer variable).

Variables come in four flavors, though they are all expanded the same and all look about the same. They are (in order of expanding scope):

- Local variables.
- Command-line variables.
- Global variables.
- Environment variables.

The classification of variables doesn't matter much, except that the classes are searched from the top (local) to the bottom (environment) when looking up a variable. The first one found wins.

2.3.1. Local Variables

Each target can have as many as seven local variables. These are variables that are only ''visible'' within that target's shell script and contain such things as the target's name, all of its sources (from all its dependency lines), those sources that were out-of-date, etc. Four local variables are defined for all targets. They are:

.TARGET
: The name of the target.

.OODATE
: The list of the sources for the target that were considered out-of-date. The order in the list is not guaranteed to be the same as the order in which the dependencies were given.

.ALLSRC
: The list of all sources for this target in the order in which they were given.

.PREFIX
: The target without its suffix and without any leading path. E.g. for the target `../../lib/compat/fsRead.c`, this variable would contain `fsRead`.

Three other local variables are set only for certain targets under special circumstances. These are the ''.IMPSRC,'' ''.ARCHIVE,'' and ''.MEMBER'' variables. When they are set and how they are used is described later.

Four of these variables may be used in sources as well as in shell scripts. These are ''.TARGET'', ''.PREFIX'', ''.ARCHIVE'' and ''.MEMBER''. The variables in the sources are expanded once for each target on the dependency line, providing what is known as a ''dynamic source,'' allowing you to specify several dependency lines at once. For example,

```
$(OBJS)         : $(.PREFIX).c
```

will create a dependency between each object file and its corresponding C source file.

2.3.2. Command-line Variables

Command-line variables are set when PMake is first invoked by giving a variable assignment as one of the arguments. For example,

```
pmake "CFLAGS = -I/sprite/src/lib/libc -O"
```

would make `CFLAGS` be a command-line variable with the given value. Any assignments to `CFLAGS` in the makefile will have no effect, because once it is set, there is (almost) nothing you can do to change a command-line variable (the search order, you see). Command-line variables may be set using any of the four assignment operators, though only = and ?= behave as you would expect them to, mostly because assignments to command-line variables are performed before the makefile is read, thus the values set in the makefile are unavailable at the time. += is the same as =, because the old value of the variable is sought only in the scope in which the assignment is taking place (for reasons of efficiency that I won't

get into here). := and ?= will work if the only variables used are in the environment. != is sort of pointless to use from the command line, since the same effect can no doubt be accomplished using the shell's own command substitution mechanisms (backquotes and all that).

2.3.3. Global Variables

Global variables are those set or appended-to in the makefile. There are two classes of global variables: those you set and those PMake sets. As I said before, the ones you set can have any name you want them to have, except they may not contain a colon or an exclamation point. The variables PMake sets (almost) always begin with a period and always contain upper-case letters, only. The variables are as follows:

.PMAKE
: The name by which PMake was invoked is stored in this variable. For compatibility, the name is also stored in the MAKE variable.

.MAKEFLAGS
: All the relevant flags with which PMake was invoked. This does not include such things as **–f** or variable assignments. Again for compatibility, this value is stored in the MFLAGS variable as well.

Two other variables, ".INCLUDES" and ".LIBS," are covered in the section on special targets in chapter 3.

Global variables may be deleted using lines of the form:

`#undef` *variable*

The '#' must be the first character on the line. Note that this may only be done on global variables.

2.3.4. Environment Variables

Environment variables are passed by the shell that invoked PMake and are given by PMake to each shell it invokes. They are expanded like any other variable, but they cannot be altered in any way.

One special environment variable, `PMAKE`, is examined by PMake for command-line flags, variable assignments, etc., it should always use. This variable is examined before the actual arguments to PMake are. In addition, all flags given to PMake, either through the `PMAKE` variable or on the command line, are placed in this environment variable and exported to each shell PMake executes. Thus recursive invocations of PMake automatically receive the same flags as the top-most one.

Using all these variables, you can compress the sample makefile even more:

```
OBJS            = a.o b.o c.o
program         : $(OBJS)
        cc $(.ALLSRC) -o $(.TARGET)
$(OBJS)         : defs.h
a.o             : a.c
        cc -c a.c
b.o             : b.c
        cc -c b.c
c.o             : c.c
        cc -c c.c
```

2.4. Comments

Comments in a makefile start with a '#' character and extend to the end of the line. They may appear anywhere you want them, except in a shell command (though the shell will treat it as a comment, too). If, for some reason, you need to use the '#' in a variable or on a dependency line, put a backslash in front of it. PMake will compress the two into a single '#' (Note: this isn't true if PMake is operating in full-compatibility mode).

NOTE

2.5. Parallelism

PMake was specifically designed to re-create several targets at once, when possible. You do not have to do anything special to cause this to happen (unless PMake was configured to not act in parallel, in which case you will have to make use of the **–L** and **–J** flags (see below)), but you do have to be careful at times.

There are several problems you are likely to encounter. One is that some makefiles (and programs) are written in such a way that it is impossible for two targets to be made at once. The program `xstr`, for example, always modifies the files `strings` and `x.c`. There is no way to change it. Thus you cannot run two of them at once without something being trashed. Similarly, if you have commands in the makefile that always send output to the same file, you will not be able to make more than one target at once unless you change the file you use. You can, for instance, add a `$$$$` to the end of the file name to tack on the process ID of the shell executing the command (each `$$` expands to a single `$`, thus giving you the shell variable `$$`). Since only one shell is used for all the commands, you'll get the same file name for each command in the script.

The other problem comes from improperly-specified dependencies that worked in Make because of its sequential, depth-first way of examining them. While I don't want to go into depth on how PMake works (look in chapter 4 if you're interested), I will warn you that files in two different "levels" of the dependency tree may be examined in a different order in PMake than they were in Make. For example, given the makefile

```
a               : b c
b               : d
```

PMake will examine the targets in the order `c`, `d`, `b`, `a`. If the makefile's author expected PMake to abort before making `c` if an error occurred while making `b`, or if `b` needed to exist before `c` was made, s/he will be sorely disappointed. The dependencies are incomplete, since in both these cases, `c` would depend on `b`. So watch out.

Another problem you may face is that, while PMake is set up to handle the output from multiple jobs in a graceful fashion, the same is not so for input. It has no way to regulate input to different jobs, so if you use the redirection from `/dev/tty` I mentioned earlier, you must be careful not to run two of the jobs at once.

2.6. Writing and Debugging a Makefile

Now you know most of what's in a makefile, what do you do next? There are two choices: (1) use one of the uncommonly-available makefile generators or (2) write your own makefile (I leave out the third choice of ignoring PMake and doing everything by hand as being beyond the bounds of common sense).

When faced with the writing of a makefile, it is usually best to start from first principles: just what *are* you trying to do? What do you want the makefile finally to produce?

To begin with a somewhat traditional example, let's say you need to write a makefile to create a program, `expr`, that takes standard infix expressions and converts them to prefix form (for no readily apparent reason). You've got three source files, in C, that make up the program: `main.c`, `parse.c`, and `output.c`. Harking back to my pithy advice about dependency lines, you write the first line of the file:

```
expr            : main.o parse.o output.o
```

because you remember `expr` is made from `.o` files, not `.c` files. Similarly for the `.o` files you produce the lines:

```
main.o          : main.c
parse.o         : parse.c
output.o        : output.c
main.o parse.o output.o : defs.h
```

Great. You've now got the dependencies specified. What you need now is commands. These commands, remember, must produce the target on the dependency line, usually by using the sources you've listed. You remember about local variables? Good, so it should come to you as no surprise when you write

```
expr            : main.o parse.o output.o
        cc -o $(.TARGET) $(.ALLSRC)
```

Why use the variables? If your program grows to produce postfix expressions too (which, of course, requires a name change or two), it is one fewer place you have to change the file. You cannot do this for the object files, however, because they depend on their corresponding source files *and* `defs.h`, thus if you said

```
cc -c $(.ALLSRC)
```

you'd get (for `main.o`):

```
cc -c main.c defs.h
```

which is wrong. So you round out the makefile with these lines:

```
main.o          : main.c
        cc -c main.c
parse.o         : parse.c
        cc -c parse.c
output.o        : output.c
        cc -c output.c
```

The makefile is now complete and will, in fact, create the program you want it to without unnecessary compilations or excessive typing on your part. There are two things wrong with it, however (aside from it being altogether too long, something I'll address in chapter 3):

1) The string "`main.o parse.o output.o`" is repeated twice, necessitating two changes when you add postfix (you were planning on that, weren't you?). This is in direct violation of de Boor's First Rule of writing makefiles:

   Anything that needs to be written more than once should be placed in a variable.

   I cannot emphasize this enough as being very important to the maintenance of a makefile and its program.

2) There is no way to alter the way compilations are performed short of editing the makefile and making the change in all places. This is evil and violates de Boor's Second Rule, which follows directly from the first:

   Any flags or programs used inside a makefile should be placed in a variable so they may be changed, temporarily or permanently, with the greatest ease.

The makefile should more properly read:

```
OBJS            = main.o parse.o output.o
expr            : $(OBJS)
        $(CC) $(CFLAGS) -o $(.TARGET) $(.ALLSRC)
main.o          : main.c
        $(CC) $(CFLAGS) -c main.c
parse.o         : parse.c
        $(CC) $(CFLAGS) -c parse.c
output.o        : output.c
        $(CC) $(CFLAGS) -c output.c
$(OBJS)         : defs.h
```

Alternatively, if you like the idea of dynamic sources mentioned in section 2.3.1, you could write it like this:

```
OBJS            = main.o parse.o output.o
expr            : $(OBJS)
        $(CC) $(CFLAGS) -o $(.TARGET) $(.ALLSRC)
$(OBJS)         : $(.PREFIX).c defs.h
        $(CC) $(CFLAGS) -c $(.PREFIX).c
```

These two rules and examples lead to de Boor's First Corollary:

Variables are your friends.

Once you've written the makefile comes the sometimes-difficult task of making sure the darn thing works. Your most helpful tool to make sure the makefile is at least syntactically correct is the **–n** flag, which allows you to see if PMake will choke on the makefile. The second thing the **–n** flag lets you do is see what PMake would do without it actually doing it, thus you can make sure the right commands would be executed were you to give PMake its head.

When you find your makefile isn't behaving as you hoped, the first question that comes to mind (after "What time is it, anyway?") is "Why not?" In answering this, two flags will serve you well: "`-d m`" and "`-p 2`." The first causes PMake to tell you as it examines each target in the makefile and indicate why it is deciding whatever it is deciding. You can then use the information printed for other targets to see where you went wrong. The "`-p 2`" flag makes PMake print out its internal state when it is done, allowing you to see that you forgot to make that one chapter depend on that file of macros you just got a new version of. The output from "`-p 2`" is intended to resemble closely a real makefile, but with additional information provided and with variables expanded in those commands PMake actually printed or executed.

Something to be especially careful about is circular dependencies. E.g.

```
a       : b
b       : c d
d       : a
```

In this case, because of how PMake works, `c` is the only thing PMake will examine, because `d` and `a` will effectively fall off the edge of the universe, making it impossible to examine `b` (or them, for that matter). PMake will tell you (if run in its normal mode) all the targets involved in any cycle it looked at (i.e. if you have two cycles in the graph (naughty, naughty), but only try to make a target in one of them, PMake will only tell you about that one. You'll have to try to make the other to find the second cycle). When run as Make, it will only print the first target in the cycle.

2.7. Invoking PMake

PMake comes with a wide variety of flags to choose from. They may appear in any order, interspersed with command-line variable assignments and targets to create. The flags are as follows:

–d *what*

This causes PMake to spew out debugging information that may prove useful to you. If you can't figure out why PMake is doing what it's doing, you might try using this flag. The *what* parameter is a string of single characters that tell PMake what aspects you are interested in. Most of what I describe will make little sense to you, unless you've dealt with Make before. Just remember where this table is and come back to it as you read on. The characters and the information they produce are as follows:

a Archive searching and caching.

c Conditional evaluation.

d The searching and caching of directories.

j Various snippets of information related to the running of the multiple shells. Not particularly interesting.

m The making of each target: what target is being examined; when it was last modified; whether it is out-of-date; etc.

p Makefile parsing.

r Remote execution.

s The application of suffix-transformation rules. (See chapter 3)

t The maintenance of the list of targets.

v Variable assignment.

Of these all, the `m` and `s` letters will be most useful to you. If the **–d** is the final argument or the argument from which it would get these key letters (see below for a note about which argument would be used) begins with a **–**, all of these debugging flags will be set, resulting in massive amounts of output.

–f *makefile*
Specify a makefile to read different from the standard makefiles (`Makefile` or `makefile`). If *makefile* is "–", PMake uses the standard input. This is useful for making quick and dirty makefiles...

–h Prints out a summary of the various flags PMake accepts. It can also be used to find out what level of concurrency was compiled into the version of PMake you are using (look at **–J** and **–L**) and various other information on how PMake was configured.

–i If you give this flag, PMake will ignore non-zero status returned by any of its shells. It's like placing a '–' before all the commands in the makefile.

–k This is similar to **–i** in that it allows PMake to continue when it sees an error, but unlike **–i**, where PMake continues blithely as if nothing went wrong, **–k** causes it to recognize the error and only continue work on those things that don't depend on the target, either directly or indirectly (through depending on something that depends on it), whose creation returned the error. The 'k' is for "keep going"...

–l PMake has the ability to lock a directory against other people executing it in the same directory (by means of a file called "LOCK.make" that it creates and checks for in the directory). This is a Good Thing because two people doing the same thing in the same place can be disastrous for the final product (too many cooks and all that). Whether this locking is the default is up to your system administrator. If locking is on, **–l** will turn it off, and vice versa. Note that this locking will not prevent *you* from invoking PMake twice in the same place — if you own the lock file, PMake will warn you about it but continue to execute.

–n This flag tells PMake not to execute the commands needed to update the out-of-date targets in the makefile. Rather, PMake will simply print the commands it would have executed and exit. This is particularly useful for checking the correctness of a makefile. If PMake doesn't do what you expect it to, it's a good chance the makefile is wrong.

–p *number*
This causes PMake to print its input in a reasonable form, though not necessarily one that would make immediate sense to anyone but me. The *number* is a bitwise-or of 1 and 2 where 1 means it should print the input before doing any processing and 2 says it should print it after everything has been re-created. Thus `-p 3` would print it twice—once before processing and once after (you might find the difference between the two interesting). This is mostly useful to me, but you may find it informative in some bizarre circumstances.

–q If you give PMake this flag, it will not try to re-create anything. It will just see if anything is out-of-date and exit non-zero if so.

–r When PMake starts up, it reads a default makefile that tells it what sort of system it's on and gives it some idea of what to do if you don't tell it anything. I'll tell you about it in chapter 3. If you give this flag, PMake won't read the default makefile.

–s This causes PMake to not print commands before they're executed. It is the equivalent of putting an '@' before every command in the makefile.

–t Rather than try to re-create a target, PMake will simply "touch" it so as to make it appear up-to-date. If the target didn't exist before, it will when PMake finishes, but if the target did exist, it

will appear to have been updated.

–v This is a mixed-compatibility flag intended to mimic the System V version of Make. It is the same as giving **–B**, and **–V** as well as turning off directory locking. Targets can still be created in parallel, however. This is the mode PMake will enter if it is invoked either as ‘‘`smake`’’ or ‘‘`vmake`’’.

–x This tells PMake it’s ok to export jobs to other machines, if they’re available. It is used when running in Make mode, as exporting in this mode tends to make things run slower than if the commands were just executed locally.

–B Forces PMake to be as backwards-compatible with Make as possible while still being itself. This includes:

- Executing one shell per shell command
- Expanding anything that looks even vaguely like a variable, with the empty string replacing any variable PMake doesn’t know.
- Refusing to allow you to escape a ‘#’ with a backslash.
- Permitting undefined variables on dependency lines and conditionals (see below). Normally this causes PMake to abort.

–C This nullifies any and all compatibility mode flags you may have given or implied up to the time the **–C** is encountered. It is useful mostly in a makefile that you wrote for PMake to avoid bad things happening when someone runs PMake as ‘‘`make`’’ or has things set in the environment that tell it to be compatible. **–C** is *not* placed in the `PMAKE` environment variable or the `.MAKEFLAGS` or `MFLAGS` global variables.

–D *variable*
Allows you to define a variable to have ‘‘`1`’’ as its value. The variable is a global variable, not a command-line variable. This is useful mostly for people who are used to the C compiler arguments and those using conditionals, which I’ll get into in section 4.3

–I *directory*
Tells PMake another place to search for included makefiles. Yet another thing to be explained in chapter 3 (section 3.2, to be precise).

–J *number*
Gives the absolute maximum number of targets to create at once on both local and remote machines.

–L *number*
This specifies the maximum number of targets to create on the local machine at once. This may be 0, though you should be wary of doing this, as PMake may hang until a remote machine becomes available, if one is not available when it is started.

–M This is the flag that provides absolute, complete, full compatibility with Make. It still allows you to use all but a few of the features of PMake, but it is non-parallel. This is the mode PMake enters if you call it ‘‘`make`.’’

–P When creating targets in parallel, several shells are executing at once, each wanting to write its own two cent’s-worth to the screen. This output must be captured by PMake in some way in order to prevent the screen from being filled with garbage even more indecipherable than you usually see. PMake has two ways of doing this, one of which provides for much cleaner output and a clear separation between the output of different jobs, the other of which provides a more immediate response so one can tell what is really happpening. The former is done by notifying you when the creation of a target starts, capturing the output and transferring it to the screen all at once when the job finishes. The latter is done by catching the output of the shell (and its children) and buffering it until an entire line is received, then printing that line preceded by an indication of which job produced the output. Since I prefer this second method, it is the one used by default. The first method will be used if you give the **–P** flag to PMake.

–V As mentioned before, the **–V** flag tells PMake to use Make’s style of expanding variables, substituting the empty string for any variable it doesn’t know.

–W There are several times when PMake will print a message at you that is only a warning, i.e. it can continue to work in spite of your having done something silly (such as forgotten a leading tab for a shell command). Sometimes you are well aware of silly things you have done and would like PMake to stop bothering you. This flag tells it to shut up about anything non-fatal.

–X This flag causes PMake to not attempt to export any jobs to another machine.

Several flags may follow a single '–'. Those flags that require arguments take them from successive parameters. E.g.

```
pmake -fDnI server.mk DEBUG /chip2/X/server/include
```

will cause PMake to read `server.mk` as the input makefile, define the variable `DEBUG` as a global variable and look for included makefiles in the directory `/chip2/X/server/include`.

2.8. Summary

A makefile is made of four types of lines:

- Dependency lines
- Creation commands
- Variable assignments
- Comments, include statements and conditional directives

A dependency line is a list of one or more targets, an operator ('`:`', '`::`', or '`!`'), and a list of zero or more sources. Sources may contain wildcards and certain local variables.

A creation command is a regular shell command preceded by a tab. In addition, if the first two characters after the tab (and other whitespace) are a combination of '`@`' or '–', PMake will cause the command to not be printed (if the character is '`@`') or errors from it to be ignored (if '–'). A blank line, dependency line or variable assignment terminates a creation script. There may be only one creation script for each target with a '`:`' or '`!`' operator.

Variables are places to store text. They may be unconditionally assigned-to using the '=' operator, appended-to using the '+=' operator, conditionally (if the variable is undefined) assigned-to with the '`?=`' operator, and assigned-to with variable expansion with the '`:=`' operator. The output of a shell command may be assigned to a variable using the '`!=`' operator. Variables may be expanded (their value inserted) by enclosing their name in parentheses or curly braces, prceeded by a dollar sign. A dollar sign may be escaped with another dollar sign. Variables are not expanded if PMake doesn't know about them. There are seven local variables: `.TARGET`, `.ALLSRC`, `.OODATE`, `.PREFIX`, `.IMPSRC`, `.ARCHIVE`, and `.MEMBER`. Four of them (`.TARGET`, `.PREFIX`, `.ARCHIVE`, and `.MEMBER`) may be used to specify "dynamic sources." Variables are good. Know them. Love them. Live them.

Debugging of makefiles is best accomplished using the **–n**, **–d m**, and **–p 2** flags.

2.9. Exercises

TBA

3. Short-cuts and Other Nice Things

Based on what I've told you so far, you may have gotten the impression that PMake is just a way of storing away commands and making sure you don't forget to compile something. Good. That's just what it is. However, the ways I've described have been inelegant, at best, and painful, at worst. This chapter contains things that make the writing of makefiles easier and the makefiles themselves shorter and easier to modify (and, occasionally, simpler). In this chapter, I assume you are somewhat more familiar with Sprite (or UNIX, if that's what you're using) than I did in chapter 2, just so you're on your toes. So without further ado...

3.1. Transformation Rules

As you know, a file's name consists of two parts: a base name, which gives some hint as to the contents of the file, and a suffix, which usually indicates the format of the file. Over the years, as UNIX† has developed, naming conventions, with regard to suffixes, have also developed that have become almost as incontrovertible as Law. E.g. a file ending in `.c` is assumed to contain C source code; one with a `.o` suffix is assumed to be a compiled, relocatable object file that may be linked into any program; a file with a `.ms` suffix is usually a text file to be processed by Troff with the –ms macro package, and so on. One of the best aspects of both Make and PMake comes from their understanding of how the suffix of a file pertains to its contents and their ability to do things with a file based soley on its suffix. This ability comes from something known as a transformation rule. A transformation rule specifies how to change a file with one suffix into a file with another suffix.

A transformation rule looks much like a dependency line, except the target is made of two known suffixes stuck together. Suffixes are made known to PMake by placing them as sources on a dependency line whose target is the special target `.SUFFIXES`. E.g.

```
.SUFFIXES       : .o .c
.c.o            :
        $(CC) $(CFLAGS) -c $(.IMPSRC)
```

The creation script attached to the target is used to transform a file with the first suffix (in this case, `.c`) into a file with the second suffix (here, `.o`). In addition, the target inherits whatever attributes have been applied to the transformation rule. The simple rule given above says that to transform a C source file into an object file, you compile it using `cc` with the `-c` flag. This rule is taken straight from the system makefile. Many transformation rules (and suffixes) are defined there, and I refer you to it for more examples (type "`pmake -h`" to find out where it is).

There are several things to note about the transformation rule given above:

1) The `.IMPSRC` variable. This variable is set to the "implied source" (the file from which the target is being created; the one with the first suffix), which, in this case, is the .c file.

2) The `CFLAGS` variable. Almost all of the transformation rules in the system makefile are set up using variables that you can alter in your makefile to tailor the rule to your needs. In this case, if you want all your C files to be compiled with the **–g** flag, to provide information for `dbx`, you would set the `CFLAGS` variable to contain `-g` ("`CFLAGS = -g`") and PMake would take care of the rest.

To give you a quick example, the makefile in 2.3.4 could be changed to this:

```
OBJS            = a.o b.o c.o
program         : $(OBJS)
        $(CC) -o $(.TARGET) $(.ALLSRC)
$(OBJS)         : defs.h
```

The transformation rule I gave above takes the place of the 6 lines[1]

```
a.o             : a.c
        cc -c a.c
b.o             : b.c
        cc -c b.c
c.o             : c.c
        cc -c c.c
```

Now you may be wondering about the dependency between the `.o` and `.c` files — it's not mentioned anywhere in the new makefile. This is because it isn't needed: one of the effects of applying a transformation rule is the target comes to depend on the implied source. That's why it's called the implied

† UNIX is a registered trademark of The Open Group in the U.S. and other countries.

[1] This is also somewhat cleaner, I think, than the dynamic source solution presented in 2.6

source .

For a more detailed example. Say you have a makefile like this:

```
a.out           : a.o b.o
        $(CC) $(.ALLSRC)
```

and a directory set up like this:

```
total 4
-rw-rw-r--  1 deboor         34 Sep  7 00:43 Makefile
-rw-rw-r--  1 deboor        119 Oct  3 19:39 a.c
-rw-rw-r--  1 deboor        201 Sep  7 00:43 a.o
-rw-rw-r--  1 deboor         69 Sep  7 00:43 b.c
```

While just typing "`pmake`" will do the right thing, it's much more informative to type "`pmake -d s`". This will show you what PMake is up to as it processes the files. In this case, PMake prints the following:

```
Suff_FindDeps (a.out)
    using existing source a.o
    applying .o -> .out to "a.o"
Suff_FindDeps (a.o)
    trying a.c...got it
    applying .c -> .o to "a.c"
Suff_FindDeps (b.o)
    trying b.c...got it
    applying .c -> .o to "b.c"
Suff_FindDeps (a.c)
    trying a.y...not there
    trying a.l...not there
    trying a.c,v...not there
    trying a.y,v...not there
    trying a.l,v...not there
Suff_FindDeps (b.c)
    trying b.y...not there
    trying b.l...not there
    trying b.c,v...not there
    trying b.y,v...not there
    trying b.l,v...not there
--- a.o ---
cc  -c a.c
--- b.o ---
cc  -c b.c
--- a.out ---
cc a.o b.o
```

`Suff_FindDeps` is the name of a function in PMake that is called to check for implied sources for a target using transformation rules. The transformations it tries are, naturally enough, limited to the ones that have been defined (a transformation may be defined multiple times, by the way, but only the most recent one will be used). You will notice, however, that there is a definite order to the suffixes that are tried. This order is set by the relative positions of the suffixes on the `.SUFFIXES` line — the earlier a suffix appears, the earlier it is checked as the source of a transformation. Once a suffix has been defined, the only way to change its position in the pecking order is to remove all the suffixes (by having a `.SUFFIXES` dependency line with no sources) and redefine them in the order you want. (Previously-defined transformation rules will be automatically redefined as the suffixes they involve are re-entered.)

Another way to affect the search order is to make the dependency explicit. In the above example, `a.out` depends on `a.o` and `b.o`. Since a transformation exists from `.o` to `.out`, PMake uses that, as

indicated by the "`using existing source a.o`" message.

The search for a transformation starts from the suffix of the target and continues through all the defined transformations, in the order dictated by the suffix ranking, until an existing file with the same base (the target name minus the suffix and any leading directories) is found. At that point, one or more transformation rules will have been found to change the one existing file into the target.

For example, ignoring what's in the system makefile for now, say you have a makefile like this:

```
.SUFFIXES       : .out .o .c .y .l
.l.c            :
        lex $(.IMPSRC)
        mv lex.yy.c $(.TARGET)
.y.c            :
        yacc $(.IMPSRC)
        mv y.tab.c $(.TARGET)
.c.o            :
        cc -c $(.IMPSRC)
.o.out          :
        cc -o $(.TARGET) $(.IMPSRC)
```

and the single file `jive.l`. If you were to type "`pmake -rd ms jive.out`," you would get the following output for `jive.out`:

```
Suff_FindDeps (jive.out)
    trying jive.o...not there
    trying jive.c...not there
    trying jive.y...not there
    trying jive.l...got it
    applying .l -> .c to "jive.l"
    applying .c -> .o to "jive.c"
    applying .o -> .out to "jive.o"
```

and this is why: PMake starts with the target `jive.out`, figures out its suffix (`.out`) and looks for things it can transform to a `.out` file. In this case, it only finds `.o`, so it looks for the file `jive.o`. It fails to find it, so it looks for transformations into a `.o` file. Again it has only one choice: `.c`. So it looks for `jive.c` and, as you know, fails to find it. At this point it has two choices: it can create the `.c` file from either a `.y` file or a `.l` file. Since `.y` came first on the `.SUFFIXES` line, it checks for `jive.y` first, but can't find it, so it looks for `jive.l` and, lo and behold, there it is. At this point, it has defined a transformation path as follows: `.l` → `.c` → `.o` → `.out` and applies the transformation rules accordingly. For completeness, and to give you a better idea of what PMake actually did with this three-step transformation, this is what PMake printed for the rest of the process:

```
Suff_FindDeps (jive.o)
    using existing source jive.c
    applying .c -> .o to "jive.c"
Suff_FindDeps (jive.c)
    using existing source jive.l
    applying .l -> .c to "jive.l"
Suff_FindDeps (jive.l)
Examining jive.l...modified 17:16:01 Oct 4, 1987...up-to-date
Examining jive.c...non-existent...out-of-date
--- jive.c ---
lex jive.l
... meaningless lex output deleted ...
mv lex.yy.c jive.c
Examining jive.o...non-existent...out-of-date
--- jive.o ---
cc -c jive.c
Examining jive.out...non-existent...out-of-date
--- jive.out ---
cc -o jive.out jive.o
```

One final question remains: what does PMake do with targets that have no known suffix? PMake simply pretends it actually has a known suffix and searches for transformations accordingly. The suffix it chooses is the source for the `.NULL` target mentioned later. In the system makefile, `.out` is chosen as the ''null suffix'' because most people use PMake to create programs. You are, however, free and welcome to change it to a suffix of your own choosing. The null suffix is ignored, however, when PMake is in compatibility mode (see chapter 4).

3.2. Including Other Makefiles

Just as for programs, it is often useful to extract certain parts of a makefile into another file and just include it in other makefiles somehow. Many compilers allow you say something like

```
#include "defs.h"
```

to include the contents of `defs.h` in the source file. PMake allows you to do the same thing for makefiles, with the added ability to use variables in the filenames. An include directive in a makefile looks either like this:

```
#include <file>
```

or this

```
#include "file"
```

The difference between the two is where PMake searches for the file: the first way, PMake will look for the file only in the system makefile directory (to find out what that directory is, give PMake the **–h** flag). For files in double-quotes, the search is more complex:

1) The directory of the makefile that's including the file.
2) The current directory (the one in which you invoked PMake).
3) The directories given by you using **–I** flags, in the order in which you gave them.
4) Directories given by `.PATH` dependency lines (see chapter 4).
5) The system makefile directory.

in that order.

You are free to use PMake variables in the filename—PMake will expand them before searching for the file. You must specify the searching method with either angle brackets or double-quotes *outside* of a variable expansion. I.e. the following

```
SYSTEM   = <command.mk>

#include $(SYSTEM)
```

won't work.

3.3. Saving Commands

There may come a time when you will want to save certain commands to be executed when everything else is done. For instance: you're making several different libraries at one time and you want to create the members in parallel. Problem is, `ranlib` is another one of those programs that can't be run more than once in the same directory at the same time (each one creates a file called `__.SYMDEF` into which it stuffs information for the linker to use. Two of them running at once will overwrite each other's file and the result will be garbage for both parties). You might want a way to save the ranlib commands til the end so they can be run one after the other, thus keeping them from trashing each other's file. PMake allows you to do this by inserting an ellipsis ("...") as a command between commands to be run at once and those to be run later.

So for the `ranlib` case above, you might do this:

```
lib1.a          : $(LIB1OBJS)
        rm -f $(.TARGET)
        ar cr $(.TARGET) $(.ALLSRC)
        ...
        ranlib $(.TARGET)

lib2.a          : $(LIB2OBJS)
        rm -f $(.TARGET)
        ar cr $(.TARGET) $(.ALLSRC)
        ...
        ranlib $(.TARGET)
```

This would save both

```
ranlib $(.TARGET)
```

commands until the end, when they would run one after the other (using the correct value for the `.TARGET` variable, of course).

Commands saved in this manner are only executed if PMake manages to re-create everything without an error.

3.4. Target Attributes

PMake allows you to give attributes to targets by means of special sources. Like everything else PMake uses, these sources begin with a period and are made up of all upper-case letters. There are various reasons for using them, and I will try to give examples for most of them. Others you'll have to find uses for yourself. Think of it as "an exercise for the reader." By placing one (or more) of these as a source on a dependency line, you are "marking the target(s) with that attribute." That's just the way I phrase it, so you know.

Any attributes given as sources for a transformation rule are applied to the target of the transformation rule when the rule is applied.

.DONTCARE If a target is marked with this attribute and PMake can't figure out how to create it, it will ignore this fact and assume the file isn't really needed or actually exists and PMake just can't find it. This may prove wrong, but the error will be noted later on, not when PMake tries to create the target so marked. This attribute also prevents PMake from attempting to touch the target if it is given the **–t** flag.

.EXEC This attribute causes its shell script to be executed while having no effect on targets that depend on it. This makes the target into a sort of subroutine. An example. Say you have some LISP files that need to be compiled and loaded into a LISP process. To do this, you echo LISP commands into a file and execute a LISP with this file as its input when everything's done. Say also that you have to load other files from another system before you can compile your files and further, that you don't want to go through the loading and dumping unless one of *your* files has changed. Your makefile might look a little bit like this (remember, this is an educational example, and don't worry about the `COMPILE` rule, all will soon become clear, grasshopper):

```
system          : init a.fasl b.fasl c.fasl
        for i in $(.ALLSRC);
        do
                echo -n '(load "' >> input
                echo -n ${i} >> input
                echo '")' >> input
        done
        echo '(dump "$(.TARGET)")' >> input
        lisp < input

a.fasl          : a.l init COMPILE
b.fasl          : b.l init COMPILE
c.fasl          : c.l init COMPILE
COMPILE         : .USE
        echo '(compile "$(.ALLSRC)")' >> input
init            : .EXEC
        echo '(load-system)' > input
```

`.EXEC` sources, don't appear in the local variables of targets that depend on them (nor are they touched if PMake is given the **–t** flag). Note that all the rules, not just that for `system`, include `init` as a source. This is because none of the other targets can be made until `init` has been made, thus they depend on it.

.EXPORT This is used to mark those targets whose creation should be sent to another machine if at all possible. This may be used by some exportation schemes if the exportation is expensive. You should ask your system administrator if it is necessary.

.EXPORTSAME
Tells the export system that the job should be exported to a machine of the same architecture as the current one. Certain operations (e.g. running text through `nroff`) can be performed the same on any architecture (CPU and operating system type), while others (e.g. compiling a program with `cc`) must be performed on a machine with the same architecture. Not all export systems will support this attribute.

.IGNORE Giving a target the `.IGNORE` attribute causes PMake to ignore errors from any of the target's commands, as if they all had '–' before them.

.INVISIBLE This allows you to specify one target as a source for another without the one affecting the other's local variables. Useful if, say, you have a makefile that creates two programs, one of which is used to create the other, so it must exist before the other is created. You could say

```
prog1           : $(PROG1OBJS) prog2 MAKEINSTALL
prog2           : $(PROG2OBJS) .INVISIBLE MAKEINSTALL
```

where `MAKEINSTALL` is some complex .USE rule (see below) that depends on the `.ALLSRC` variable containing the right things. Without the `.INVISIBLE` attribute for `prog2`, the `MAKEINSTALL` rule couldn't be applied. This is not as useful as it should be, and the semantics may change (or the whole thing go away) in the not-too-distant

future.

.JOIN This is another way to avoid performing some operations in parallel while permitting everything else to be done so. Specifically it forces the target's shell script to be executed only if one or more of the sources was out-of-date. In addition, the target's name, in both its `.TARGET` variable and all the local variables of any target that depends on it, is replaced by the value of its `.ALLSRC` variable. As an example, suppose you have a program that has four libraries that compile in the same directory along with, and at the same time as, the program. You again have the problem with `ranlib` that I mentioned earlier, only this time it's more severe: you can't just put the ranlib off to the end since the program will need those libraries before it can be re-created. You can do something like this:

```
program         : $(OBJS) libraries
        cc -o $(.TARGET) $(.ALLSRC)

libraries       : lib1.a lib2.a lib3.a lib4.a .JOIN
        ranlib $(.OODATE)
```

In this case, PMake will re-create the `$(OBJS)` as necessary, along with `lib1.a`, `lib2.a`, `lib3.a` and `lib4.a`. It will then execute `ranlib` on any library that was changed and set `program`'s `.ALLSRC` variable to contain what's in `$(OBJS)` followed by "`lib1.a lib2.a lib3.a lib4.a`." In case you're wondering, it's called `.JOIN` because it joins together different threads of the "input graph" at the target marked with the attribute. Another aspect of the .JOIN attribute is it keeps the target from being created if the **–t** flag was given.

.MAKE The `.MAKE` attribute marks its target as being a recursive invocation of PMake. This forces PMake to execute the script associated with the target (if it's out-of-date) even if you gave the **–n** or **–t** flag. By doing this, you can start at the top of a system and type

```
pmake -n
```

and have it descend the directory tree (if your makefiles are set up correctly), printing what it would have executed if you hadn't included the **–n** flag.

.NOEXPORT If possible, PMake will attempt to export the creation of all targets to another machine (this depends on how PMake was configured). Sometimes, the creation is so simple, it is pointless to send it to another machine. If you give the target the `.NOEXPORT` attribute, it will be run locally, even if you've given PMake the **–L 0** flag.

.NOTMAIN Normally, if you do not specify a target to make in any other way, PMake will take the first target on the first dependency line of a makefile as the target to create. That target is known as the "Main Target" and is labeled as such if you print the dependencies out using the **–p** flag. Giving a target this attribute tells PMake that the target is definitely *not* the Main Target. This allows you to place targets in an included makefile and have PMake create something else by default.

.PRECIOUS When PMake is interrupted (you type control-C at the keyboard), it will attempt to clean up after itself by removing any half-made targets. If a target has the `.PRECIOUS` attribute, however, PMake will leave it alone. An additional side effect of the '::' operator is to mark the targets as `.PRECIOUS`.

.SILENT Marking a target with this attribute keeps its commands from being printed when they're executed, just as if they had an '@' in front of them.

.USE By giving a target this attribute, you turn it into PMake's equivalent of a macro. When the target is used as a source for another target, the other target acquires the commands, sources and attributes (except `.USE`) of the source. If the target already has commands, the `.USE` target's commands are added to the end. If more than one .USE-marked source is given to a target, the rules are applied sequentially.

The typical .USE rule (as I call them) will use the sources of the target to which it is applied (as stored in the .ALLSRC variable for the target) as its "arguments," if you will. For example, you probably noticed that the commands for creating lib1.a and lib2.a in the example in section 3.3 were exactly the same. You can use the .USE attribute to eliminate the repetition, like so:

```
lib1.a          : $(LIB1OBJS) MAKELIB
lib2.a          : $(LIB2OBJS) MAKELIB

MAKELIB         : .USE
        rm -f $(.TARGET)
        ar cr $(.TARGET) $(.ALLSRC)
        ...
        ranlib $(.TARGET)
```

Several system makefiles (not to be confused with The System Makefile) make use of these .USE rules to make your life easier (they're in the default, system makefile directory...take a look). Note that the .USE rule source itself (MAKELIB) does not appear in any of the targets's local variables. There is no limit to the number of times I could use the MAKELIB rule. If there were more libraries, I could continue with "lib3.a : $(LIB3OBJS) MAKELIB" and so on and so forth.

3.5. Special Targets

As there were in Make, so there are certain targets that have special meaning to PMake. When you use one on a dependency line, it is the only target that may appear on the left-hand-side of the operator. As for the attributes and variables, all the special targets begin with a period and consist of upper-case letters only. I won't describe them all in detail because some of them are rather complex and I'll describe them in more detail than you'll want in chapter 4. The targets are as follows:

.BEGIN Any commands attached to this target are executed before anything else is done. You can use it for any initialization that needs doing.

.DEFAULT This is sort of a .USE rule for any target (that was used only as a source) that PMake can't figure out any other way to create. It's only "sort of" a .USE rule because only the shell script attached to the .DEFAULT target is used. The .IMPSRC variable of a target that inherits .DEFAULT's commands is set to the target's own name.

.END This serves a function similar to .BEGIN, in that commands attached to it are executed once everything has been re-created (so long as no errors occurred). It also serves the extra function of being a place on which PMake can hang commands you put off to the end. Thus the script for this target will be executed before any of the commands you save with the "...".

.EXPORT The sources for this target are passed to the exportation system compiled into PMake. Some systems will use these sources to configure themselves. You should ask your system administrator about this.

.IGNORE This target marks each of its sources with the .IGNORE attribute. If you don't give it any sources, then it is like giving the **–i** flag when you invoke PMake — errors are ignored for all commands.

.INCLUDES The sources for this target are taken to be suffixes that indicate a file that can be included in a program source file. The suffix must have already been declared with .SUFFIXES (see below). Any suffix so marked will have the directories on its search path (see .PATH, below) placed in the .INCLUDES variable, each preceded by a **–I** flag. This variable can then be used as an argument for the compiler in the normal fashion. The .h suffix is already marked in this way in the system makefile. E.g. if you have

```
.SUFFIXES       : .bitmap
.PATH.bitmap    : /usr/local/X/lib/bitmaps
.INCLUDES       : .bitmap
```

PMake will place "`-I/usr/local/X/lib/bitmaps`" in the `.INCLUDES` variable and you can then say

```
cc $(.INCLUDES) -c xprogram.c
```

(Note: the `.INCLUDES` variable is not actually filled in until the entire makefile has been read.)

.INTERRUPT
When PMake is interrupted, it will execute the commands in the script for this target, if it exists.

.LIBS This does for libraries what `.INCLUDES` does for include files, except the flag used is **–L**, as required by those linkers that allow you to tell them where to find libraries. The variable used is `.LIBS`. Be forewarned that PMake may not have been compiled to do this if the linker on your system doesn't accept the **–L** flag, though the `.LIBS` variable will always be defined once the makefile has been read.

.MAIN If you didn't give a target (or targets) to create when you invoked PMake, it will take the sources of this target as the targets to create.

.MAKEFLAGS
This target provides a way for you to always specify flags for PMake when the makefile is used. The flags are just as they would be typed to the shell (except you can't use shell variables unless they're in the environment), though the **–f** and **–r** flags have no effect.

.NULL This allows you to specify what suffix PMake should pretend a file has if, in fact, it has no known suffix. Only one suffix may be so designated. The last source on the dependency line is the suffix that is used (you should, however, only give one suffix...).

.PATH If you give sources for this target, PMake will take them as directories in which to search for files it cannot find in the current directory. If you give no sources, it will clear out any directories added to the search path before. Since the effects of this all get very complex, I'll leave it til chapter four to give you a complete explanation.

.PATH*suffix* This does a similar thing to `.PATH`, but it does it only for files with the given suffix. The suffix must have been defined already. Look at **Search Paths** (section 4.1) for more information.

.PRECIOUS Similar to `.IGNORE`, this gives the `.PRECIOUS` attribute to each source on the dependency line, unless there are no sources, in which case the `.PRECIOUS` attribute is given to every target in the file.

.RECURSIVE
This target applies the `.MAKE` attribute to all its sources. It does nothing if you don't give it any sources.

.SHELL PMake is not constrained to only using the Bourne shell to execute the commands you put in the makefile. You can tell it some other shell to use with this target. Check out **A Shell is a Shell is a Shell** (section 4.4) for more information.

.SILENT When you use `.SILENT` as a target, it applies the `.SILENT` attribute to each of its sources. If there are no sources on the dependency line, then it is as if you gave PMake the **–s** flag and no commands will be echoed.

.SUFFIXES This is used to give new file suffixes for PMake to handle. Each source is a suffix PMake should recognize. If you give a `.SUFFIXES` dependency line with no sources, PMake will forget about all the suffixes it knew (this also nukes the null suffix). For those targets that need to have suffixes defined, this is how you do it.

In addition to these targets, a line of the form

attribute **:** *sources*

applies the *attribute* to all the targets listed as *sources*.

3.6. Modifying Variable Expansion

Variables need not always be expanded verbatim. PMake defines several modifiers that may be applied to a variable's value before it is expanded. You apply a modifier by placing it after the variable name with a colon between the two, like so:

`$`{*VARIABLE*`:`*modifier*}

Each modifier is a single character followed by something specific to the modifier itself. You may apply as many modifiers as you want — each one is applied to the result of the previous and is separated from the previous by another colon.

There are seven ways to modify a variable's expansion, most of which come from the C shell variable modification characters:

M*pattern*
: This is used to select only those words (a word is a series of characters that are neither spaces nor tabs) that match the given *pattern*. The pattern is a wildcard pattern like that used by the shell, where `*` means 0 or more characters of any sort; `?` is any single character; `[abcd]` matches any single character that is either 'a', 'b', 'c' or 'd' (there may be any number of characters between the brackets); `[0-9]` matches any single character that is between '0' and '9' (i.e. any digit. This form may be freely mixed with the other bracket form), and '\' is used to escape any of the characters '*', '?', '[' or ':', leaving them as regular characters to match themselves in a word. For example, the system makefile `<makedepend.mk>` uses "`$(CFLAGS:M-[ID]*)`" to extract all the `-I` and `-D` flags that would be passed to the C compiler. This allows it to properly locate include files and generate the correct dependencies.

N*pattern*
: This is identical to `:M` except it substitutes all words that don't match the given pattern.

S/*search-string*/*replacement-string*/[g]
: Causes the first occurrence of *search-string* in the variable to be replaced by *replacement-string*, unless the `g` flag is given at the end, in which case all occurences of the string are replaced. The substitution is performed on each word in the variable in turn. If *search-string* begins with a `^`, the string must match starting at the beginning of the word. If *search-string* ends with a `$`, the string must match to the end of the word (these two may be combined to force an exact match). If a backslash preceeds these two characters, however, they lose their special meaning. Variable expansion also occurs in the normal fashion inside both the *search-string* and the *replacement-string*, **except** that a backslash is used to prevent the expansion of a `$`, not another dollar sign, as is usual. Note that *search-string* is just a string, not a pattern, so none of the usual regular-expression/wildcard characters have any special meaning save `^` and `$`. In the replacement string, the `&` character is replaced by the *search-string* unless it is preceded by a backslash. You are allowed to use any character except colon or exclamation point to separate the two strings. This so-called delimiter character may be placed in either string by preceeding it with a backslash.

T
: Replaces each word in the variable expansion by its last component (its "tail"). For example, given

```
OBJS = ../lib/a.o b /usr/lib/libm.a
TAILS = $(OBJS:T)
```

the variable `TAILS` would expand to "`a.o b libm.a`."

H This is similar to `:T`, except that every word is replaced by everything but the tail (the "head"). Using the same definition of `OBJS`, the string "`$(OBJS:H)`" would expand to "`../lib /usr/lib`." Note that the final slash on the heads is removed and anything without a head is replaced by the empty string.

E `:E` replaces each word by its suffix ("extension"). So "`$(OBJS:E)`" would give you "`.o .a`."

R This replaces each word by everything but the suffix (the "root" of the word). "`$(OBJS:R)`" expands to " `../lib/a b /usr/lib/libm`."

In addition, the System V style of substitution is also supported. This looks like:

$(*VARIABLE*:*search-string*=*replacement*)

It must be the last modifier in the chain. The search is anchored at the end of each word, so only suffixes or whole words may be replaced.

3.7. More on Debugging

3.8. More Exercises

(3.1) You've got a set programs, each of which is created from its own assembly-language source file (suffix `.asm`). Each program can be assembled into two versions, one with error-checking code assembled in and one without. You could assemble them into files with different suffixes (`.eobj` and `.obj`, for instance), but your linker only understands files that end in `.obj`. To top it all off, the final executables *must* have the suffix `.exe`. How can you still use transformation rules to make your life easier (Hint: assume the error-checking versions have `ec` tacked onto their prefix)?

(3.2) Assume, for a moment or two, you want to perform a sort of "indirection" by placing the name of a variable into another one, then you want to get the value of the first by expanding the second somehow. Unfortunately, PMake doesn't allow constructs like

```
$($(FOO))
```

What do you do? Hint: no further variable expansion is performed after modifiers are applied, thus if you cause a $ to occur in the expansion, that's what will be in the result.

4. PMake for Gods

This chapter is devoted to those facilities in PMake that allow you to do a great deal in a makefile with very little work, as well as do some things you couldn't do in Make without a great deal of work (and perhaps the use of other programs). The problem with these features, is they must be handled with care, or you will end up with a mess.

Once more, I assume a greater familiarity with UNIX or Sprite than I did in the previous two chapters.

4.1. Search Paths

PMake supports the dispersal of files into multiple directories by allowing you to specify places to look for sources with `.PATH` targets in the makefile. The directories you give as sources for these targets make up a "search path." Only those files used exclusively as sources are actually sought on a search path, the assumption being that anything listed as a target in the makefile can be created by the makefile and thus should be in the current directory.

There are two types of search paths in PMake: one is used for all types of files (including included makefiles) and is specified with a plain `.PATH` target (e.g. "`.PATH : RCS`"), while the other is specific to a certain type of file, as indicated by the file's suffix. A specific search path is indicated by immediately following the `.PATH` with the suffix of the file. For instance

```
.PATH.h          : /sprite/lib/include /sprite/att/lib/include
```

would tell PMake to look in the directories `/sprite/lib/include` and `/sprite/att/lib/include` for any files whose suffix is `.h`.

The current directory is always consulted first to see if a file exists. Only if it cannot be found there are the directories in the specific search path, followed by those in the general search path, consulted.

A search path is also used when expanding wildcard characters. If the pattern has a recognizable suffix on it, the path for that suffix will be used for the expansion. Otherwise the default search path is employed.

When a file is found in some directory other than the current one, all local variables that would have contained the target's name (`.ALLSRC`, and `.IMPSRC`) will instead contain the path to the file, as found by PMake. Thus if you have a file `../lib/mumble.c` and a makefile

```
.PATH.c         : ../lib
mumble          : mumble.c
        $(CC) -o $(.TARGET) $(.ALLSRC)
```

the command executed to create `mumble` would be "`cc -o mumble ../lib/mumble.c`." (As an aside, the command in this case isn't strictly necessary, since it will be found using transformation rules if it isn't given. This is because `.out` is the null suffix by default and a transformation exists from `.c` to `.out`. Just thought I'd throw that in.)

If a file exists in two directories on the same search path, the file in the first directory on the path will be the one PMake uses. So if you have a large system spread over many directories, it would behoove you to follow a naming convention that avoids such conflicts.

Something you should know about the way search paths are implemented is that each directory is read, and its contents cached, exactly once — when it is first encountered — so any changes to the directories while PMake is running will not be noted when searching for implicit sources, nor will they be found when PMake attempts to discover when the file was last modified, unless the file was created in the current directory. While people have suggested that PMake should read the directories each time, my experience suggests that the caching seldom causes problems. In addition, not caching the directories slows things down enormously because of PMake's attempts to apply transformation rules through non-existent files — the number of extra file-system searches is truly staggering, especially if many files without suffixes are used and the null suffix isn't changed from `.out`.

4.2. Archives and Libraries

UNIX and Sprite allow you to merge files into an archive using the `ar` command. Further, if the files are relocatable object files, you can run `ranlib` on the archive and get yourself a library that you can link into any program you want. The main problem with archives is they double the space you need to store the archived files, since there's one copy in the archive and one copy out by itself. The problem with libraries is you usually think of them as `-lm` rather than `/usr/lib/libm.a` and the linker thinks they're out-of-date if you so much as look at them.

PMake solves the problem with archives by allowing you to tell it to examine the files in the archives (so you can remove the individual files without having to regenerate them later). To handle the problem with libraries, PMake adds an additional way of deciding if a library is out-of-date:

- If the table of contents is older than the library, or is missing, the library is out-of-date.

A library is any target that looks like "`-l`name" or that ends in a suffix that was marked as a library using the `.LIBS` target. `.a` is so marked in the system makefile.

Members of an archive are specified as "*archive*(*member*[*member...*])". Thus "'`libdix.a(window.o)`" specifies the file `window.o` in the archive `libdix.a`. You may also use wildcards to specify the members of the archive. Just remember that most the wildcard characters will only find *existing* files.

A file that is a member of an archive is treated specially. If the file doesn't exist, but it is in the archive, the modification time recorded in the archive is used for the file when determining if the file is out-of-date. When figuring out how to make an archived member target (not the file itself, but the file in the archive — the *archive*(*member*) target), special care is taken with the transformation rules, as follows:

- *archive*(*member*) is made to depend on *member*.
- The transformation from the *member*'s suffix to the *archive*'s suffix is applied to the *archive*(*member*) target.
- The *archive*(*member*)'s `.TARGET` variable is set to the name of the *member* if *member* is actually a target, or the path to the member file if *member* is only a source.
- The `.ARCHIVE` variable for the *archive*(*member*) target is set to the name of the *archive*.
- The `.MEMBER` variable is set to the actual string inside the parentheses. In most cases, this will be the same as the `.TARGET` variable.
- The *archive*(*member*)'s place in the local variables of the targets that depend on it is taken by the value of its `.TARGET` variable.

Thus, a program library could be created with the following makefile:

```
.o.a            :
        ...
        rm -f $(.TARGET:T)
OBJS            = obj1.o obj2.o obj3.o
libprog.a       : libprog.a($(OBJS))
        ar cru $(.TARGET) $(.OODATE)
        ranlib $(.TARGET)
```

This will cause the three object files to be compiled (if the corresponding source files were modified after the object file or, if that doesn't exist, the archived object file), the out-of-date ones archived in `libprog.a`, a table of contents placed in the archive and the newly-archived object files to be removed.

All this is used in the `makelib.mk` system makefile to create a single library with ease. This makefile looks like this:

```
#
# Rules for making libraries. The object files that make up the library are
# removed once they are archived.
#
# To make several libararies in parallel, you should define the variable
# "many_libraries". This will serialize the invocations of ranlib.
#
# To use, do something like this:
#
# OBJECTS = <files in the library>
#
# fish.a: fish.a($(OBJECTS)) MAKELIB
#
#

#ifndef _MAKELIB_MK
_MAKELIB_MK     =

#include  <po.mk>

.po.a .o.a :
     ...
     rm -f $(.MEMBER)

ARFLAGS         ?= crl

#
# Re-archive the out-of-date members and recreate the library's table of
# contents using ranlib. If many_libraries is defined, put the ranlib off
# til the end so many libraries can be made at once.
#
MAKELIB         : .USE .PRECIOUS
     ar $(ARFLAGS) $(.TARGET) $(.OODATE)
#ifndef no_ranlib
# ifdef many_libraries
     ...
# endif many_libraries
     ranlib $(.TARGET)
#endif no_ranlib

#endif _MAKELIB_MK
```

4.3. On the Condition...

Like the C compiler before it, PMake allows you to configure the makefile, based on the current environment, using conditional statements. A conditional looks like this:

`#if` *boolean expression*
lines
`#elif` *another boolean expression*
more lines
`#else`
still more lines
`#endif`

They may be nested to a maximum depth of 30 and may occur anywhere (except in a comment, of course). The "`#`" must the very first character on the line.

Each *boolean expression* is made up of terms that look like function calls, the standard C boolean operators `&&`, `||`, and `!`, and the standard relational operators ==, `!=`, >, >=, <, and <=, with == and `!=` being overloaded to allow string comparisons as well. `&&` represents logical AND; `||` is logical OR and `!` is logical NOT. The arithmetic and string operators take precedence over all three of these operators, while NOT takes precedence over AND, which takes precedence over OR. This precedence may be overridden with parentheses, and an expression may be parenthesized to your heart's content. Each term looks like a call on one of four functions:

make — The syntax is `make(`*target*`)` where *target* is a target in the makefile. This is true if the given target was specified on the command line, or as the source for a `.MAIN` target (note that the sources for `.MAIN` are only used if no targets were given on the command line).

defined — The syntax is `defined(`*variable*`)` and is true if *variable* is defined. Certain variables are defined in the system makefile that identify the system on which PMake is being run.

exists — The syntax is `exists(`*file*`)` and is true if the file can be found on the global search path (i.e. that defined by `.PATH` targets, not by `.PATH`*suffix* targets).

empty — This syntax is much like the others, except the string inside the parentheses is of the same form as you would put between parentheses when expanding a variable, complete with modifiers and everything. The function returns true if the resulting string is empty (NOTE: an undefined variable in this context will cause at the very least a warning message about a malformed conditional, and at the worst will cause the process to stop once it has read the makefile. If you want to check for a variable being defined or empty, use the expression "`!defined(`*var*`) || empty(`*var*`)`" as the definition of `||` will prevent the `empty()` from being evaluated and causing an error, if the variable is undefined). This can be used to see if a variable contains a given word, for example:

```
#if !empty(var:Mword)
```

The arithmetic and string operators may only be used to test the value of a variable. The lefthand side must contain the variable expansion, while the righthand side contains either a string, enclosed in double-quotes, or a number. The standard C numeric conventions (except for specifying an octal number) apply to both sides. E.g.

```
#if $(OS) == 4.3

#if $(MACHINE) == "sun3"

#if $(LOAD_ADDR) < 0xc000
```

are all valid conditionals. In addition, the numeric value of a variable can be tested as a boolean as follows:

```
#if $(LOAD)
```

would see if `LOAD` contains a non-zero value and

```
#if !$(LOAD)
```

would test if `LOAD` contains a zero value.

In addition to the bare "`#if`," there are other forms that apply one of the first two functions to each term. They are as follows:

`ifdef`	defined
`ifndef`	!defined
`ifmake`	make
`ifnmake`	!make

There are also the ‘‘else if’’ forms: `elif`, `elifdef`, `elifndef`, `elifmake`, and `elifnmake`.

For instance, if you wish to create two versions of a program, one of which is optimized (the production version) and the other of which is for debugging (has symbols for dbx), you have two choices: you can create two makefiles, one of which uses the `-g` flag for the compilation, while the other uses the `-O` flag, or you can use another target (call it `debug`) to create the debug version. The construct below will take care of this for you. I have also made it so defining the variable `DEBUG` (say with `pmake -D DEBUG`) will also cause the debug version to be made.

```
#if defined(DEBUG) || make(debug)
CFLAGS       += -g
#else
CFLAGS       += -O
#endif
```

There are, of course, problems with this approach. The most glaring annoyance is that if you want to go from making a debug version to making a production version, you have to remove all the object files, or you will get some optimized and some debug versions in the same program. Another annoyance is you have to be careful not to make two targets that ‘‘conflict’’ because of some conditionals in the makefile. For instance

```
#if make(print)
FORMATTER    = ditroff -Plaser_printer
#endif
#if make(draft)
FORMATTER    = nroff -Pdot_matrix_printer
#endif
```

would wreak havok if you tried ‘‘`pmake draft print`’’ since you would use the same formatter for each target. As I said, this all gets somewhat complicated.

4.4. A Shell is a Shell is a Shell

In normal operation, the Bourne Shell (better known as ‘‘`sh`’’) is used to execute the commands to re-create targets. PMake also allows you to specify a different shell for it to use when executing these commands. There are several things PMake must know about the shell you wish to use. These things are specified as the sources for the `.SHELL` target by keyword, as follows:

path=*path*
: PMake needs to know where the shell actually resides, so it can execute it. If you specify this and nothing else, PMake will use the last component of the path and look in its table of the shells it knows and use the specification it finds, if any. Use this if you just want to use a different version of the Bourne or C Shell (yes, PMake knows how to use the C Shell too).

name=*name*
: This is the name by which the shell is to be known. It is a single word and, if no other keywords are specified (other than **path**), it is the name by which PMake attempts to find a specification for it (as mentioned above). You can use this if you would just rather use the C Shell than the Bourne Shell (‘‘`.SHELL: name=csh`’’ will do it).

quiet=*echo-off command*
: As mentioned before, PMake actually controls whether commands are printed by introducing commands into the shell’s input stream. This keyword, and the next two, control what those commands are. The **quiet** keyword is the command used to turn echoing off. Once it is turned off, echoing is expected to remain off until the echo-on command is given.

echo=*echo-on command*
: The command PMake should give to turn echoing back on again.

filter=*printed echo-off command*
: Many shells will echo the echo-off command when it is given. This keyword tells PMake in what

format the shell actually prints the echo-off command. Wherever PMake sees this string in the shell's output, it will delete it and any following whitespace, up to and including the next newline. See the example at the end of this section for more details.

echoFlag=*flag to turn echoing on*
Unless a target has been marked `.SILENT`, PMake wants to start the shell running with echoing on. To do this, it passes this flag to the shell as one of its arguments. If either this or the next flag begins with a '–', the flags will be passed to the shell as separate arguments. Otherwise, the two will be concatenated (if they are used at the same time, of course).

errFlag=*flag to turn error checking on*
Likewise, unless a target is marked `.IGNORE`, PMake wishes error-checking to be on from the very start. To this end, it will pass this flag to the shell as an argument. The same rules for an initial '–' apply as for the **echoFlag**.

check=*command to turn error checking on*
Just as for echo-control, error-control is achieved by inserting commands into the shell's input stream. This is the command to make the shell check for errors. It also serves another purpose if the shell doesn't have error-control as commands, but I'll get into that in a minute. Again, once error checking has been turned on, it is expected to remain on until it is turned off again.

ignore=*command to turn error checking off*
This is the command PMake uses to turn error checking off. It has another use if the shell doesn't do error-control, but I'll tell you about that... now.

hasErrCtl=*yes or no*
This takes a value that is either **yes** or **no**. Now you might think that the existence of the **check** and **ignore** keywords would be enough to tell PMake if the shell can do error-control, but you'd be wrong. If **hasErrCtl** is **yes**, PMake uses the check and ignore commands in a straight-forward manner. If this is **no**, however, their use is rather different. In this case, the check command is used as a template, in which the string **%s** is replaced by the command that's about to be executed, to produce a command for the shell that will echo the command to be executed. The ignore command is also used as a template, again with **%s** replaced by the command to be executed, to produce a command that will execute the command to be executed and ignore any error it returns. When these strings are used as templates, you must provide newline(s) ("`\n`") in the appropriate place(s).

The strings that follow these keywords may be enclosed in single or double quotes (the quotes will be stripped off) and may contain the usual C backslash-characters (\n is newline, \r is return, \b is backspace, \' escapes a single-quote inside single-quotes, \" escapes a double-quote inside double-quotes). Now for an example.

This is actually the contents of the `<shx.mk>` system makefile, and causes PMake to use the Bourne Shell in such a way that each command is printed as it is executed. That is, if more than one command is given on a line, each will be printed separately. Similarly, each time the body of a loop is executed, the commands within that loop will be printed, etc. The specification runs like this:

```
#
# This is a shell specification to have the bourne shell echo
# the commands just before executing them, rather than when it reads
# them. Useful if you want to see how variables are being expanded, etc.
#
.SHELL  : path=/bin/sh \
    quiet="set -" \
    echo="set -x" \
    filter="+ set - " \
    echoFlag=x \
    errFlag=e \
    hasErrCtl=yes \
    check="set -e" \
    ignore="set +e"
```

It tells PMake the following:

- The shell is located in the file `/bin/sh`. It need not tell PMake that the name of the shell is `sh` as PMake can figure that out for itself (it's the last component of the path).
- The command to stop echoing is `set -`.
- The command to start echoing is `set -x`.
- When the echo off command is executed, the shell will print `+ set -` (The '+' comes from using the `-x` flag (rather than the `-v` flag PMake usually uses)). PMake will remove all occurences of this string from the output, so you don't notice extra commands you didn't put there.
- The flag the Bourne Shell will take to start echoing in this way is the `-x` flag. The Bourne Shell will only take its flag arguments concatenated as its first argument, so neither this nor the **errFlag** specification begins with a –.
- The flag to use to turn error-checking on from the start is `-e`.
- The shell can turn error-checking on and off, and the commands to do so are `set +e` and `set -e`, respectively.

I should note that this specification is for Bourne Shells that are not part of Berkeley UNIX, as shells from Berkeley don't do error control. You can get a similar effect, however, by changing the last three lines to be:

```
hasErrCtl=no \
check="echo \"+ %s\"\n" \
ignore="sh -c '%s || exit 0\n"
```

This will cause PMake to execute the two commands

```
echo "+ cmd"
sh -c 'cmd || true'
```

for each command for which errors are to be ignored. (In case you are wondering, the thing for `ignore` tells the shell to execute another shell without error checking on and always exit 0, since the `||` causes the `exit 0` to be executed only if the first command exited non-zero, and if the first command exited zero, the shell will also exit zero, since that's the last command it executed).

4.5. Compatibility

There are three (well, 3 ½) levels of backwards-compatibility built into PMake. Most makefiles will need none at all. Some may need a little bit of work to operate correctly when run in parallel. Each level encompasses the previous levels (e.g. **–B** (one shell per command) implies **–V**) The three levels are described in the following three sections.

4.5.1. DEFCON 3 — Variable Expansion

As noted before, PMake will not expand a variable unless it knows of a value for it. This can cause problems for makefiles that expect to leave variables undefined except in special circumstances (e.g. if more flags need to be passed to the C compiler or the output from a text processor should be sent to a different printer). If the variables are enclosed in curly braces ("`${PRINTER}`"), the shell will let them pass. If they are enclosed in parentheses, however, the shell will declare a syntax error and the make will come to a grinding halt.

You have two choices: change the makefile to define the variables (their values can be overridden on the command line, since that's where they would have been set if you used Make, anyway) or always give the **–V** flag (this can be done with the `.MAKEFLAGS` target, if you want).

4.5.2. DEFCON 2 — The Number of the Beast

Then there are the makefiles that expect certain commands, such as changing to a different directory, to not affect other commands in a target's creation script. You can solve this is either by going back to executing one shell per command (which is what the **–B** flag forces PMake to do), which slows the process down a good bit and requires you to use semicolons and escaped newlines for shell constructs, or by changing the makefile to execute the offending command(s) in a subshell (by placing the line inside parentheses), like so:

```
install :: .MAKE
    (cd src; $(.PMAKE) install)
    (cd lib; $(.PMAKE) install)
    (cd man; $(.PMAKE) install)
```

This will always execute the three makes (even if the **–n** flag was given) because of the combination of the "::" operator and the `.MAKE` attribute. Each command will change to the proper directory to perform the install, leaving the main shell in the directory in which it started.

4.5.3. DEFCON 1 — Imitation is the Not the Highest Form of Flattery

The final category of makefile is the one where every command requires input, the dependencies are incompletely specified, or you simply cannot create more than one target at a time, as mentioned earlier. In addition, you may not have the time or desire to upgrade the makefile to run smoothly with PMake. If you are the conservative sort, this is the compatibility mode for you. It is entered either by giving PMake the **–M** flag (for Make), or by executing PMake as "`make`." In either case, PMake performs things exactly like Make (while still supporting most of the nice new features PMake provides). This includes:

- No parallel execution.
- Targets are made in the exact order specified by the makefile. The sources for each target are made in strict left-to-right order, etc.
- A single Bourne shell is used to execute each command, thus the shell's `$$` variable is useless, changing directories doesn't work across command lines, etc.
- If no special characters exist in a command line, PMake will break the command into words itself and execute the command directly, without executing a shell first. The characters that cause PMake to execute a shell are: `#`, `=`, `|`, `^`, `(`, `)`, `{`, `}`, `;`, `&`, `<`, `>`, `*`, `?`, `[`, `]`, `:`, `$`, `` ` ``, and `\`. You should notice that these are all the characters that are given special meaning by the shell (except `'` and `,` which PMake deals with all by its lonesome).
- The use of the null suffix is turned off.

4.6. The Way Things Work

When PMake reads the makefile, it parses sources and targets into nodes in a graph. The graph is directed only in the sense that PMake knows which way is up. Each node contains not only links to all its parents and children (the nodes that depend on it and those on which it depends, respectively), but

also a count of the number of its children that have already been processed.

The most important thing to know about how PMake uses this graph is that the traversal is breadth-first and occurs in two passes.

After PMake has parsed the makefile, it begins with the nodes the user has told it to make (either on the command line, or via a `.MAIN` target, or by the target being the first in the file not labeled with the `.NOTMAIN` attribute) placed in a queue. It continues to take the node off the front of the queue, mark it as something that needs to be made, pass the node to `Suff_FindDeps` (mentioned earlier) to find any implicit sources for the node, and place all the node's children that have yet to be marked at the end of the queue. If any of the children is a `.USE` rule, its attributes are applied to the parent, then its commands are appended to the parent's list of commands and its children are linked to its parent. The parent's unmade children counter is then decremented (since the `.USE` node has been processed). You will note that this allows a `.USE` node to have children that are `.USE` nodes and the rules will be applied in sequence. If the node has no children, it is placed at the end of another queue to be examined in the second pass. This process continues until the first queue is empty.

At this point, all the leaves of the graph are in the examination queue. PMake removes the node at the head of the queue and sees if it is out-of-date. If it is, it is passed to a function that will execute the commands for the node asynchronously. When the commands have completed, all the node's parents have their unmade children counter decremented and, if the counter is then 0, they are placed on the examination queue. Likewise, if the node is up-to-date. Only those parents that were marked on the downward pass are processed in this way. Thus PMake traverses the graph back up to the nodes the user instructed it to create. When the examination queue is empty and no shells are running to create a target, PMake is finished.

Once all targets have been processed, PMake executes the commands attached to the `.END` target, either explicitly or through the use of an ellipsis in a shell script. If there were no errors during the entire process but there are still some targets unmade (PMake keeps a running count of how many targets are left to be made), there is a cycle in the graph. PMake does a depth-first traversal of the graph to find all the targets that weren't made and prints them out one by one.

5. Answers to Exercises

(3.1) This is something of a trick question, for which I apologize. The trick comes from the UNIX definition of a suffix, which PMake doesn't necessarily share. You will have noticed that all the suffixes used in this tutorial (and in UNIX in general) begin with a period (`.ms`, `.c`, etc.). Now, PMake's idea of a suffix is more like English's: it's the characters at the end of a word. With this in mind, one possible solution to this problem goes as follows:

```
.SUFFIXES       : ec.exe .exe ec.obj .obj .asm
ec.objec.exe .obj.exe :
        link -o $(.TARGET) $(.IMPSRC)
.asmec.obj      :
        asm -o $(.TARGET) -DDO_ERROR_CHECKING $(.IMPSRC)
.asm.obj        :
        asm -o $(.TARGET) $(.IMPSRC)
```

(3.2) The trick to this one lies in the "`:=`" variable-assignment operator and the "`:S`" variable-expansion modifier. Basically what you want is to take the pointer variable, so to speak, and transform it into an invocation of the variable at which it points. You might try something like

```
$(PTR:S/^/\$(/:S/$/))
```

which places "`$(`" at the front of the variable name and "`)`" at the end, thus transforming "`VAR`," for example, into "`$(VAR)`," which is just what we want. Unfortunately (as you know if you've tried it), since, as it says in the hint, PMake does no further substitution on the result of a modified expansion, that's *all* you get. The solution is to make use of "`:=`" to place that string into yet another variable, then invoke the other variable directly:

```
*PTR            := $(PTR:S/^/\$(/:S/$/)/)
```

You can then use "`$(*PTR)`" to your heart's content.

6. Glossary of Jargon

attribute: A property given to a target that causes PMake to treat it differently.

command script: The lines immediately following a dependency line that specify commands to execute to create each of the targets on the dependency line. Each line in the command script must begin with a tab.

command-line variable: A variable defined in an argument when PMake is first executed. Overrides all assignments to the same variable name in the makefile.

conditional: A construct much like that used in C that allows a makefile to be configured on the fly based on the local environment, or on what is being made by that invocation of PMake.

creation script: Commands used to create a target. See "command script."

dependency: The relationship between a source and a target. This comes in three flavors, as indicated by the operator between the target and the source. ':' gives a straight time-wise dependency (if the target is older than the source, the target is out-of-date), while '!' provides simply an ordering and always considers the target out-of-date. '::' is much like ':', save it creates multiple instances of a target each of which depends on its own list of sources.

dynamic source: This refers to a source that has a local variable invocation in it. It allows a single dependency line to specify a different source for each target on the line.

global variable: Any variable defined in a makefile. Takes precedence over variables defined in the environment, but not over command-line or local variables.

input graph: What PMake constructs from a makefile. Consists of nodes made of the targets in the makefile, and the links between them (the dependencies). The links are directed (from source to target) and there may not be any cycles (loops) in the graph.

local variable: A variable defined by PMake visible only in a target's shell script. There are seven local variables, not all of which are defined for every target: `.TARGET`, `.ALLSRC`, `.OODATE`, `.PREFIX`, `.IMPSRC`, `.ARCHIVE`, and `.MEMBER`. `.TARGET`, `.PREFIX`, `.ARCHIVE`, and `.MEMBER` may be used on dependency lines to create "dynamic sources."

makefile: A file that describes how a system is built. If you don't know what it is after reading this tutorial....

modifier: A letter, following a colon, used to alter how a variable is expanded. It has no effect on the variable itself.

operator: What separates a source from a target (on a dependency line) and specifies the relationship between the two. There are three: '`:`', '`::`', and '`!`'.

search path: A list of directories in which a file should be sought. PMake's view of the contents of directories in a search path does not change once the makefile has been read. A file is sought on a search path only if it is exclusively a source.

shell: A program to which commands are passed in order to create targets.

source: Anything to the right of an operator on a dependency line. Targets on the dependency line are usually created from the sources.

special target: A target that causes PMake to do special things when it's encountered.

suffix: The tail end of a file name. Usually begins with a period, `.c` or `.ms`, e.g.

target: A word to the left of the operator on a dependency line. More generally, any file that PMake might create. A file may be (and often is) both a target and a source (what it is depends on how PMake is looking at it at the time — sort of like the wave/particle duality of light, you know).

transformation rule: A special construct in a makefile that specifies how to create a file of one type from a file of another, as indicated by their suffixes.

variable expansion: The process of substituting the value of a variable for a reference to it. Expansion may be altered by means of modifiers.

variable: A place in which to store text that may be retrieved later. Also used to define the local environment. Conditionals exist that test whether a variable is defined or not.

Table of Contents

1.	Introduction	1
2.	The Basics of PMake	1
2.1.	Dependency Lines	2
2.2.	Shell Commands	3
2.3.	Variables	5
2.3.1.	Local Variables	6
2.3.2.	Command-line Variables	6
2.3.3.	Global Variables	7
2.3.4.	Environment Variables	7
2.4.	Comments	7
2.5.	Parallelism	8
2.6.	Writing and Debugging a Makefile	8
2.7.	Invoking PMake	10
2.8.	Summary	13
2.9.	Exercises	13
3.	Short-cuts and Other Nice Things	13
3.1.	Transformation Rules	14
3.2.	Including Other Makefiles	17
3.3.	Saving Commands	18
3.4.	Target Attributes	18
3.5.	Special Targets	21
3.6.	Modifying Variable Expansion	23
3.7.	More on Debugging	24
3.8.	More Exercises	24
4.	PMake for Gods	24
4.1.	Search Paths	24
4.2.	Archives and Libraries	25
4.3.	On the Condition...	27
4.4.	A Shell is a Shell is a Shell	29
4.5.	Compatibility	31
4.5.1.	DEFCON 3 — Variable Expansion	32
4.5.2.	DEFCON 2 — The Number of the Beast	32
4.5.3.	DEFCON 1 — Imitation is the Not the Highest Form of Flattery	32
4.6.	The Way Things Work	32
5.	Answers to Exercises	33
6.	Glossary of Jargon	34

RCS—A System for Version Control

Walter F. Tichy

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907

ABSTRACT

An important problem in program development and maintenance is version control, i.e., the task of keeping a software system consisting of many versions and configurations well organized. The Revision Control System (RCS) is a software tool that assists with that task. RCS manages revisions of text documents, in particular source programs, documentation, and test data. It automates the storing, retrieval, logging and identification of revisions, and it provides selection mechanisms for composing configurations. This paper introduces basic version control concepts and discusses the practice of version control using RCS. For conserving space, RCS stores deltas, i.e., differences between successive revisions. Several delta storage methods are discussed. Usage statistics show that RCS's delta storage method is space and time efficient. The paper concludes with a detailed survey of version control tools.

Keywords: configuration management, history management, version control, revisions, deltas.

1991/01/03

RCS—A System for Version Control

Walter F. Tichy

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907

1. Introduction

Version control is the task of keeping software systems consisting of many versions and configurations well organized. The Revision Control System (RCS) is a set of UNIX commands that assist with that task.

RCS' primary function is to manage *revision groups*. A revision group is a set of text documents, called *revisions*, that evolved from each other. A new revision is created by manually editing an existing one. RCS organizes the revisions into an ancestral tree. The initial revision is the root of the tree, and the tree edges indicate from which revision a given one evolved. Besides managing individual revision groups, RCS provides flexible selection functions for composing configurations. RCS may be combined with MAKE[1], resulting in a powerful package for version control.

RCS also offers facilities for merging updates with customer modifications, for distributed software development, and for automatic identification. Identification is the 'stamping' of revisions and configurations with unique markers. These markers are akin to serial numbers, telling software maintainers unambiguously which configuration is before them.

RCS is designed for both production and experimental environments. In production environments, access controls detect update conflicts and prevent overlapping changes. In experimental environments, where strong controls are counterproductive, it is possible to loosen the controls.

Although RCS was originally intended for programs, it is useful for any text that is revised frequently and whose previous revisions must be preserved. RCS has been applied successfully to store the source text for drawings, VLSI layouts, documentation, specifications, test data, form letters and articles.

This paper discusses the practice of version control using RCS. It also introduces basic version control concepts, useful for clarifying current practice and designing similar systems. Revision groups of individual components are treated in the next three sections, and the extensions to configurations follow. Because of its size, a survey of version control tools appears at the end of the paper.

2. Getting started with RCS

Suppose a text file *f.c* is to be placed under control of RCS. Invoking the check-in command

ci f.c

creates a new revision group with the contents of *f.c* as the initial revision (numbered 1.1) and stores the group into the file *f.c,v*. Unless told otherwise, the command deletes *f.c*. It also asks for a description of the group. The description should state the common purpose of all revisions in the group, and becomes part of the group's documentation. All later check-in commands will ask for a log entry, which should summarize the changes made. (The first revision is assigned a default log message, which

An earlier version of this paper was published in *Software—Practice & Experience* **15**, 7 (July 1985), 637-654.

just records the fact that it is the initial revision.)

Files ending in *,v* are called *RCS files* (*v* stands for *v*ersions); the others are called working files. To get back the working file *f.c* in the previous example, execute the check-out command:

co f.c

This command extracts the latest revision from the revision group *f.c,v* and writes it into *f.c*. The file *f.c* can now be edited and, when finished, checked back in with *ci*:

ci f.c

Ci assigns number 1.2 to the new revision. If *ci* complains with the message

ci error: no lock set by <login>

then the system administrator has decided to configure RCS for a production environment by enabling the 'strict locking feature'. If this feature is enabled, all RCS files are initialized such that check-in operations require a lock on the previous revision (the one from which the current one evolved). Locking prevents overlapping modifications if several people work on the same file. If locking is required, the revision should have been locked during the check-out by using the option *–l*:

co –l f.c

Of course it is too late now for the check-out with locking, because *f.c* has already been changed; checking out the file again would overwrite the modifications. (To prevent accidental overwrites, *co* senses the presence of a working file and asks whether the user really intended to overwrite it. The overwriting check-out is sometimes useful for backing up to the previous revision.) To be able to proceed with the check-in in the present case, first execute

rcs –l f.c

This command retroactively locks the latest revision, unless someone else locked it in the meantime. In this case, the two programmers involved have to negotiate whose modifications should take precedence.

If an RCS file is private, i.e., if only the owner of the file is expected to deposit revisions into it, the strict locking feature is unnecessary and may be disabled. If strict locking is disabled, the owner of the RCS file need not have a lock for check-in. For safety reasons, all others still do. Turning strict locking off and on is done with the commands:

rcs –U f.c and *rcs –L f.c*

These commands enable or disable the strict locking feature for each RCS file individually. The system administrator only decides whether strict locking is enabled initially.

To reduce the clutter in a working directory, all RCS files can be moved to a subdirectory with the name *RCS*. RCS commands look first into that directory for RCS files. All the commands presented above work with the *RCS* subdirectory without change.†

It may be undesirable that *ci* deletes the working file. For instance, sometimes one would like to save the current revision, but continue editing. Invoking

ci –l f.c

checks in *f.c* as usual, but performs an additional check-out with locking afterwards. Thus, the working file does not disappear after the check-in. Similarly, the option *–u* does a check-in followed by a check-out without locking. This option is useful if the file is needed for compilation after the check-in. Both options update the identification markers in the working file (see below).

---

† Pairs of RCS and working files can actually be specified in 3 ways: a) both are given, b) only the working file is given, c) only the RCS file is given. If a pair is given, both files may have arbitrary path prefixes; RCS commands pair them up intelligently.

Besides the operations *ci* and *co*, RCS provides the following commands:

ident	extract identification markers
rcs	change RCS file attributes
rcsclean	remove unchanged working files (optional)
rcsdiff	compare revisions
rcsfreeze	record a configuration (optional)
rcsmerge	merge revisions
rlog	read log messages and other information in RCS files

A synopsis of these commands appears in the Appendix.

2.1. Automatic Identification

RCS can stamp source and object code with special identification strings, similar to product and serial numbers. To obtain such identification, place the marker

Id

into the text of a revision, for instance inside a comment. The check-out operation will replace this marker with a string of the form

$Id: filename revisionnumber date time author state locker $

This string need never be touched, because *co* keeps it up to date automatically. To propagate the marker into object code, simply put it into a literal character string. In C, this is done as follows:

```
static char rcsid[] = "$Id$";
```

The command *ident* extracts such markers from any file, in particular from object code. *Ident* helps to find out which revisions of which modules were used in a given program. It returns a complete and unambiguous component list, from which a copy of the program can be reconstructed. This facility is invaluable for program maintenance.

There are several additional identification markers, one for each component of Id. The marker

Log

has a similar function. It accumulates the log messages that are requested during check-in. Thus, one can maintain the complete history of a revision directly inside it, by enclosing it in a comment. Figure 1 is a partial reproduction of a log contained in revision 4.1 of the file *ci.c*. The log appears at the beginning of the file, and makes it easy to determine what the recent modifications were.

```
/* $Log: ci.c,v $
 * Revision 4.1  1983/05/10  17:03:06  wft
 * Added option –d and –w, and updated assignment of date, etc. to new delta.
 * Added handling of default branches.
 *
 * Revision 3.9  1983/02/15  15:25:44  wft
 * Added call to fastcopy() to copy remainder of RCS file.
 *
 * Revision 3.8  1983/01/14  15:34:05  wft
 * Added ignoring of interrupts while new RCS file is renamed;
 * avoids deletion of RCS files by interrupts.
 *
 * Revision 3.7  1982/12/10  16:09:20  wft
 * Corrected checking of return code from diff.
 * An RCS file now inherits its mode during the first ci from the working file,
 * except that write permission is removed.
 */
```

Figure 1. Log entries produced by the marker Log.

Since revisions are stored in the form of differences, each log message is physically stored once, independent of the number of revisions present. Thus, the Log marker incurs negligible space overhead.

3. The RCS Revision Tree

RCS arranges revisions in an ancestral tree. The *ci* command builds this tree; the auxiliary command *rcs* prunes it. The tree has a root revision, normally numbered 1.1, and successive revisions are numbered 1.2, 1.3, etc. The first field of a revision number is called the *release number* and the second one the *level number*. Unless given explicitly, the *ci* command assigns a new revision number by incrementing the level number of the previous revision. The release number must be incremented explicitly, using the *–r* option of *ci*. Assuming there are revisions 1.1, 1.2, and 1.3 in the RCS file f.c,v, the command

ci –r2.1 f.c or *ci –r2 f.c*

assigns the number 2.1 to the new revision. Later check-ins without the *–r* option will assign the numbers 2.2, 2.3, and so on. The release number should be incremented only at major transition points in the development, for instance when a new release of a software product has been completed.

3.1. When are branches needed?

A young revision tree is slender: It consists of only one branch, called the trunk. As the tree ages, side branches may form. Branches are needed in the following 4 situations.

Temporary fixes

Suppose a tree has 5 revisions grouped in 2 releases, as illustrated in Figure 2. Revision 1.3, the last one of release 1, is in operation at customer sites, while release 2 is in active development.

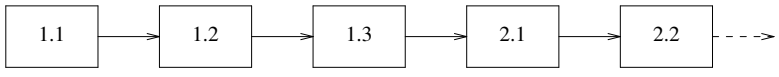


Figure 2. A slender revision tree.

Now imagine a customer requesting a fix of a problem in revision 1.3, although actual development has moved on to release 2. RCS does not permit an extra revision to be spliced in between 1.3 and 2.1, since that would not reflect the actual development history. Instead, create a branch at revision 1.3, and check in the fix on that branch. The first branch starting at 1.3 has number 1.3.1, and the revisions on that branch are numbered 1.3.1.1, 1.3.1.2, etc. The double numbering is needed to allow for another branch at 1.3, say 1.3.2. Revisions on the second branch would be

numbered 1.3.2.1, 1.3.2.2, and so on. The following steps create branch 1.3.1 and add revision 1.3.1.1:

```
co  –r1.3  f.c      — check out revision 1.3
edit  f.c           — change it
ci  –r1.3.1  f.c    — check it in on branch 1.3.1
```

This sequence of commands transforms the tree of Figure 2 into the one in Figure 3. Note that it may be necessary to incorporate the differences between 1.3 and 1.3.1.1 into a revision at level 2. The operation *rcsmerge* automates this process (see the Appendix).

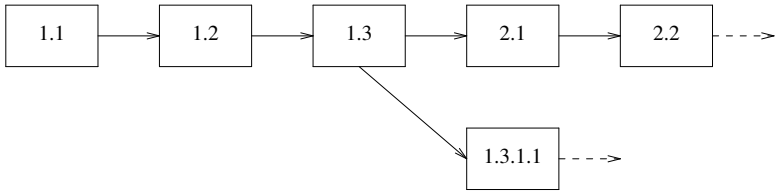


Figure 3. A revision tree with one side branch

Distributed development and customer modifications

Assume a situation as in Figure 2, where revision 1.3 is in operation at several customer sites, while release 2 is in development. Customer sites should use RCS to store the distributed software. However, customer modifications should not be placed on the same branch as the distributed source; instead, they should be placed on a side branch. When the next software distribution arrives, it should be appended to the trunk of the customer's RCS file, and the customer can then merge the local modifications back into the new release. In the above example, a customer's RCS file would contain the following tree, assuming that the customer has received revision 1.3, added his local modifications as revision 1.3.1.1, then received revision 2.4, and merged 2.4 and 1.3.1.1, resulting in 2.4.1.1.

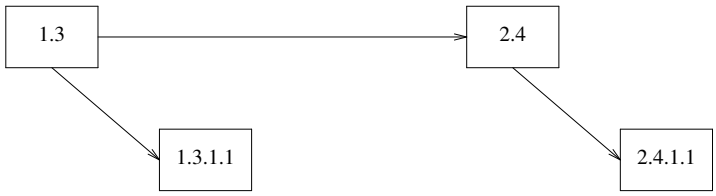


Figure 4. A customer's revision tree with local modifications.

This approach is actually practiced in the CSNET project, where several universities and a company cooperate in developing a national computer network.

Parallel development

Sometimes it is desirable to explore an alternate design or a different implementation technique in parallel with the main line development. Such development should be carried out on a side branch. The experimental changes may later be moved into the main line, or abandoned.

Conflicting updates

A common occurrence is that one programmer has checked out a revision, but cannot complete the assignment for some reason. In the meantime, another person must perform another modification immediately. In that case, the second person should check-out the same revision, modify it, and check it in on a side branch, for later merging.

Every node in a revision tree consists of the following attributes: a revision number, a check-in date and time, the author's identification, a log entry, a state and the actual text. All these attributes are determined at the time the revision is checked in. The state attribute indicates the status of a revision. It is set automatically to 'experimental' during check-in. A revision can later be promoted to a higher status, for example 'stable' or 'released'. The set of states is user-defined.

3.2. Revisions are represented as deltas

For conserving space, RCS stores revisions in the form of deltas, i.e., as differences between revisions. The user interface completely hides this fact.

A delta is a sequence of edit commands that transforms one string into another. The deltas employed by RCS are line-based, which means that the only edit commands allowed are insertion and deletion of lines. If a single character in a line is changed, the edit scripts consider the entire line changed. The program *diff*[2] produces a small, line-based delta between pairs of text files. A character-based edit script would take much longer to compute, and would not be significantly shorter.

Using deltas is a classical space-time tradeoff: deltas reduce the space consumed, but increase access time. However, a version control tool should impose as little delay as possible on programmers. Excessive delays discourage the use of version controls, or induce programmers to take shortcuts that compromise system integrity. To gain reasonably fast access time for both editing and compiling, RCS arranges deltas in the following way. The most recent revision on the trunk is stored intact. All other revisions on the trunk are stored as reverse deltas. A reverse delta describes how to go backward in the development history: it produces the desired revision if applied to the successor of that revision. This implementation has the advantage that extraction of the latest revision is a simple and fast copy operation. Adding a new revision to the trunk is also fast: *ci* simply adds the new revision intact, replaces the previous revision with a reverse delta, and keeps the rest of the old deltas. Thus, *ci* requires the computation of only one new delta.

Branches need special treatment. The naive solution would be to store complete copies for the tips of all branches. Clearly, this approach would cost too much space. Instead, RCS uses *forward* deltas for branches. Regenerating a revision on a side branch proceeds as follows. First, extract the latest revision on the trunk; secondly, apply reverse deltas until the fork revision for the branch is obtained; thirdly, apply forward deltas until the desired branch revision is reached. Figure 5 illustrates a tree with one side branch. Triangles pointing to the left and right represent reverse and forward deltas, respectively.

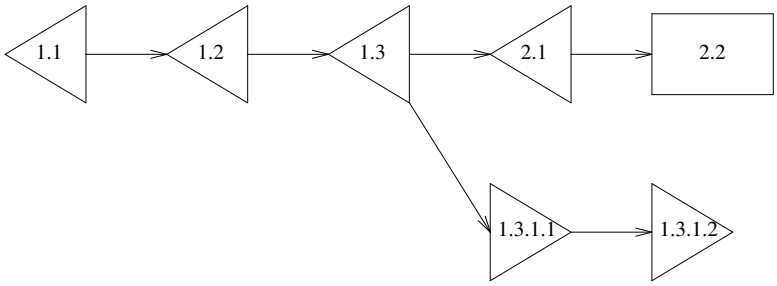


Figure 5. A revision tree with reverse and forward deltas.

Although implementing fast check-out for the latest trunk revision, this arrangement has the disadvantage that generation of other revisions takes time proportional to the number of deltas applied. For example, regenerating the branch tip in Figure 5 requires application of five deltas (including the initial one). Since usage statistics show that the latest trunk revision is the one that is retrieved in 95 per cent of all cases (see the section on usage statistics), biasing check-out time in favor of that revision results in significant savings. However, careful implementation of the delta application process is necessary to provide low retrieval overhead for other revisions, in particular for branch tips.

There are several techniques for delta application. The naive one is to pass each delta to a general-purpose text editor. A prototype of RCS invoked the UNIX editor *ed* both for applying deltas and for expanding the identification markers. Although easy to implement, performance was poor, owing to the high start-up costs and excess generality of *ed*. An intermediate version of RCS used a special-purpose, stream-oriented editor. This technique reduced the cost of applying a delta to the cost of checking out the latest trunk revision. The reason for this behavior is that each delta application involves a complete pass over the preceding revision.

However, there is a much better algorithm. Note that the deltas are line oriented and that most of the work of a stream editor involves copying unchanged lines from one revision to the next. A faster algorithm avoids unnecessary copying of character strings by using a *piece table*. A piece table is a one-dimensional array, specifying how a given revision is 'pieced together' from lines in the RCS file. Suppose piece table PT_r represents revision *r*. Then $PT_r[i]$ contains the starting position of line *i* of revision *r*. Application of the next delta transforms piece table PT_r into PT_{r+1}. For instance, a delete command removes a series of entries from the piece table. An insertion command inserts new entries, moving the entries following the insertion point further down the array. The inserted entries point to the text lines in the delta. Thus, no I/O is involved except for reading the delta itself. When all deltas have been applied to the piece table, a sequential pass through the table looks up each line in the RCS file and copies it to the output file, updating identification markers at the same time. Of course, the RCS file must permit random access, since the copied lines are scattered throughout that file.

The piece table approach has the property that the time for applying a single delta is roughly determined by the size of the delta, and not by the size of the revision. For example, if a delta is 10 per cent of the size of a revision, then applying it takes only 10 per cent of the time to generate the latest trunk revision. (The stream editor would take 100 per cent.)

There is an important alternative for representing deltas that affects performance. SCCS[3], a precursor of RCS, uses *interleaved* deltas. A file containing interleaved deltas is partitioned into blocks of lines. Each block has a header that specifies to which revision(s) the block belongs. The blocks are sorted out in such a way that a single pass over the file can pick up all the lines belonging to a given revision. Thus, the regeneration time for all revisions is the same: all headers must be inspected, and the associated blocks either copied or skipped. As the number of revisions increases, the cost of retrieving any revision is much higher than the cost of checking out the latest trunk revision with reverse deltas. A detailed comparison of SCCS's interleaved deltas and RCS's reverse deltas can be found in Reference 4. This reference considers the version of RCS with the stream editor only. The piece table method improves performance further, so that RCS is always faster than SCCS, except if 10 or more deltas are applied.

Additional speed-up for both delta methods can be obtained by caching the most recently generated revision, as has been implemented in DSEE.[5] With caching, access time to frequently used revisions can approach normal file access time, at the cost of some additional space.

4. Locking: A Controversial Issue

The locking mechanism for RCS was difficult to design. The problem and its solution are first presented in their 'pure' form, followed by a discussion of the complications caused by 'real-world' considerations.

RCS must prevent two or more persons from depositing competing changes of the same revision. Suppose two programmers check out revision 2.4 and modify it. Programmer A checks in a revision before programmer B. Unfortunately, programmer B has not seen A's changes, so the effect is that A's changes are covered up by B's deposit. A's changes are not lost since all revisions are saved, but they are confined to a single revision.‡

This conflict is prevented in RCS by locking. Whenever someone intends to edit a revision (as opposed to reading or compiling it), the revision should be checked out and locked, using the *–l* option on *co*. On subsequent check-in, *ci* tests the lock and then removes it. At most one programmer at a time may lock a particular revision, and only this programmer may check in the succeeding revision. Thus, while a revision is locked, it is the exclusive responsibility of the locker.

An important maxim for software tools like RCS is that they must not stand in the way of making progress with a project. This consideration leads to several weakenings of the locking mechanism. First of all, even if a revision is locked, it can still be checked out. This is necessary if other people wish to

‡ Note that this problem is entirely different from the atomicity problem. Atomicity means that concurrent update operations on the same RCS file cannot be permitted, because that may result in inconsistent data. Atomic updates are essential (and implemented in RCS), but do not solve the conflict discussed here.

compile or inspect the locked revision while the next one is in preparation. The only operations they cannot do are to lock the revision or to check in the succeeding one. Secondly, check-in operations on other branches in the RCS file are still possible; the locking of one revision does not affect any other revision. Thirdly, revisions are occasionally locked for a long period of time because a programmer is absent or otherwise unable to complete the assignment. If another programmer has to make a pressing change, there are the following three alternatives for making progress: a) find out who is holding the lock and ask that person to release it; b) check out the locked revision, modify it, check it in on a branch, and merge the changes later; c) break the lock. Breaking a lock leaves a highly visible trace, namely an electronic mail message that is sent automatically to the holder of the lock, recording the breaker and a commentary requested from him. Thus, breaking locks is tolerated under certain circumstances, but will not go unnoticed. Experience has shown that the automatic mail message attaches a high enough stigma to lock breaking, such that programmers break locks only in real emergencies, or when a co-worker resigns and leaves locked revisions behind.

If an RCS file is private, i.e., when a programmer owns an RCS file and does not expect anyone else to perform check-in operations, locking is an unnecessary nuisance. In this case, the 'strict locking feature' discussed earlier may be disabled, provided that file protection is set such that only the owner may write the RCS file. This has the effect that only the owner can check-in revisions, and that no lock is needed for doing so.

As added protection, each RCS file contains an access list that specifies the users who may execute update operations. If an access list is empty, only normal UNIX file protection applies. Thus, the access list is useful for restricting the set of people who would otherwise have update permission. Just as with locking, the access list has no effect on read-only operations such as *co*. This approach is consistent with the UNIX philosophy of openness, which contributes to a productive software development environment.

5. Configuration Management

The preceding sections described how RCS deals with revisions of individual components; this section discusses how to handle configurations. A configuration is a set of revisions, where each revision comes from a different revision group, and the revisions are selected according to a certain criterion. For example, in order to build a functioning compiler, the 'right' revisions from the scanner, the parser, the optimizer and the code generator must be combined. RCS, in conjunction with MAKE, provides a number of facilities to effect a smooth selection.

5.1. RCS Selection Functions

Default selection

During development, the usual selection criterion is to choose the latest revision of all components. The *co* command makes this selection by default. For example, the command

```
co  *,v
```

retrieves the latest revision on the default branch of each RCS file in the current directory. The default branch is usually the trunk, but may be set to be a side branch. Side branches as defaults are needed in distributed software development, as discussed in the section on the RCS revision tree.

Release based selection

Specifying a release or branch number selects the latest revision in that release or branch. For instance,

```
co  -r2  *,v
```

retrieves the latest revision with release number 2 from each RCS file. This selection is convenient if a release has been completed and development has moved on to the next release.

State and author based selection

If the highest level number within a given release number is not the desired one, the state attribute can help. For example,

*co –r2 –sReleased *,v*

retrieves the latest revision with release number 2 whose state attribute is 'Released'. Of course, the state attribute has to be set appropriately, using the *ci* or *rcs* commands. Another alternative is to select a revision by its author, using the *–w* option.

Date based selection

Revisions may also be selected by date. Suppose a release of an entire system was completed and current on March 4, at 1:00 p.m. local time. Then the command

*co –d'March 4, 1:00 pm LT' *,v*

checks out all the components of that release, independent of the numbering. The *–d* option specifies a 'cutoff date', i.e., the revision selected has a check-in date that is closest to, but not after the date given.

Name based selection

The most powerful selection function is based on assigning symbolic names to revisions and branches. In large systems, a single release number or date is not sufficient to collect the appropriate revisions from all groups. For example, suppose one wishes to combine release 2 of one subsystem and release 15 of another. Most likely, the creation dates of those releases differ also. Thus, a single revision number or date passed to the *co* command will not suffice to select the right revisions. Symbolic revision numbers solve this problem. Each RCS file may contain a set of symbolic names that are mapped to numeric revision numbers. For example, assume the symbol *V3* is bound to release number 2 in file *s,v*, and to revision number 15.9 in *t,v*. Then the single command

co –rV3 s,v t,v

retrieves the latest revision of release 2 from *s,v*, and revision 15.9 from *t,v*. In a large system with many modules, checking out all revisions with one command greatly simplifies configuration management.

Judicious use of symbolic revision numbers helps with organizing large configurations. A special command, *rcsfreeze*, assigns a symbolic revision number to a selected revision in every RCS file. *Rcsfreeze* effectively freezes a configuration. The assigned symbolic revision number selects all components of the configuration. If necessary, symbolic numbers may even be intermixed with numeric ones. Thus, *V3.5* in the above example would select revision 2.5 in *s,v* and branch 15.9.5 in *t,v*.

The options *–r*, *–s*, *–w* and *–d* may be combined. If a branch is given, the latest revision on that branch satisfying all conditions is retrieved; otherwise, the default branch is used.

5.2. Combining MAKE and RCS

MAKE[1] is a program that processes configurations. It is driven by configuration specifications recorded in a special file, called a 'Makefile'. MAKE avoids redundant processing steps by comparing creation dates of source and processed objects. For example, when instructed to compile all modules of a given system, it only recompiles those source modules that were changed since they were processed last.

MAKE has been extended with an auto-checkout feature for RCS.* When a certain file to be processed is not present, MAKE attempts a check-out operation. If successful, MAKE performs the required processing, and then deletes the checked out file to conserve space. The selection parameters discussed above can be passed to MAKE either as parameters, or directly embedded in the Makefile.

* This auto-checkout extension is available only in some versions of MAKE, e.g. GNU MAKE.

MAKE has also been extended to search the subdirectory named *RCS* for needed files, rather than just the current working directory. However, if a working file is present, MAKE totally ignores the corresponding RCS file and uses the working file. (In newer versions of MAKE distributed by AT&T and others, auto-checkout can be achieved with the rule DEFAULT, instead of a special extension of MAKE. However, a file checked out by the rule DEFAULT will not be deleted after processing. *Rcsclean* can be used for that purpose.)

With auto-checkout, RCS/MAKE can effect a selection rule especially tuned for multi-person software development and maintenance. In these situations, programmers should obtain configurations that consist of the revisions they have personally checked out plus the latest checked in revision of all other revision groups. This schema can be set up as follows.

Each programmer chooses a working directory and places into it a symbolic link, named *RCS*, to the directory containing the relevant RCS files. The symbolic link makes sure that *co* and *ci* operations need only specify the working files, and that the Makefile need not be changed. The programmer then checks out the needed files and modifies them. If MAKE is invoked, it composes configurations by selecting those revisions that are checked out, and the rest from the subdirectory *RCS*. The latter selection may be controlled by a symbolic revision number or any of the other selection criteria. If there are several programmers editing in separate working directories, they are insulated from each other's changes until checking in their modifications.

Similarly, a maintainer can recreate an older configuration by starting to work in an empty working directory. During the initial MAKE invocation, all revisions are selected from RCS files. As the maintainer checks out files and modifies them, a new configuration is gradually built up. Every time MAKE is invoked, it substitutes the modified revisions into the configuration being manipulated.

A final application of RCS is to use it for storing Makefiles. Revision groups of Makefiles represent multiple versions of configurations. Whenever a configuration is baselined or distributed, the best approach is to unambiguously fix the configuration with a symbolic revision number by calling *rcsfreeze*, to embed that symbol into the Makefile, and to check in the Makefile (using the same symbolic revision number). With this approach, old configurations can be regenerated easily and reliably.

6. Usage Statistics

The following usage statistics were collected on two DEC VAX-11/780 computers of the Purdue Computer Science Department. Both machines are mainly used for research purposes. Thus, the data reflect an environment in which the majority of projects involve prototyping and advanced software development, but relatively little long-term maintenance.

For the first experiment, the *ci* and *co* operations were instrumented to log the number of backward and forward deltas applied. The data were collected during a 13 month period from Dec. 1982 to Dec. 1983. Table I summarizes the results.

Operation	Total operations	Total deltas applied	Mean deltas applied	Operations with >1 delta		Branch operations	
co	7867	9320	1.18	509	(6%)	203	(3%)
ci	3468	2207	0.64	85	(2%)	75	(2%)
ci & co	11335	11527	1.02	594	(5%)	278	(2%)

Table I. Statistics for *co* and *ci* operations.

The first two lines show statistics for check-out and check-in; the third line shows the combination. Recall that *ci* performs an implicit check-out to obtain a revision for computing the delta. In all measures presented, the most recent revision (stored intact) counts as one delta. The number of deltas applied represents the number of passes necessary, where the first 'pass' is a copying step.

Note that the check-out operation is executed more than twice as frequently as the check-in operation. The fourth column gives the mean number of deltas applied in all three cases. For *ci*, the mean number of deltas applied is less than one. The reasons are that the initial check-in requires no delta at all, and that the only time *ci* requires more than one delta is for branches. Column 5 shows the actual number of operations that applied more than one delta. The last column indicates that branches were

not used often.

The last three columns demonstrate that the most recent trunk revision is by far the most frequently accessed. For RCS, check-out of this revision is a simple copy operation, which is the absolute minimum given the copy-semantics of *co*. Access to older revisions and branches is more common in non-academic environments, yet even if access to older deltas were an order of magnitude more frequent, the combined average number of deltas applied would still be below 1.2. Since RCS is faster than SCCS until up to 10 delta applications, reverse deltas are clearly the method of choice.

The second experiment, conducted in March of 1984, involved surveying the existing RCS files on our two machines. The goal was to determine the mean number of revisions per RCS file, as well as the space consumed by them. Table II shows the results. (Tables I and II were produced at different times and are unrelated.)

	Total RCS files	Total revisions	Mean revisions	Mean size of RCS files	Mean size of revisions	Overhead
All files	8033	11133	1.39	6156	5585	1.10
Files with ≥ 2 deltas	1477	4578	3.10	8074	6041	1.34

Table II. Statistics for RCS files.

The mean number of revisions per RCS file is 1.39. Columns 5 and 6 show the mean sizes (in bytes) of an RCS file and of the latest revision of each RCS file, respectively. The 'overhead' column contains the ratio of the mean sizes. Assuming that all revisions in an RCS file are approximately the same size, this ratio gives a measure of the space consumed by the extra revisions.

In our sample, over 80 per cent of the RCS files contained only a single revision. The reason is that our systems programmers routinely check in all source files on the distribution tapes, even though they may never touch them again. To get a better indication of how much space savings are possible with deltas, all measures with those files that contained 2 or more revisions were recomputed. Only for those files is RCS necessary. As shown in the second line, the average number of revisions for those files is 3.10, with an overhead of 1.34. This means that the extra 2.10 deltas require 34 per cent extra space, or 16 per cent per extra revision. Rochkind[3] measured the space consumed by SCCS, and reported an average of 5 revisions per group and an overhead of 1.37 (or about 9 per cent per extra revision). In a later paper, Glasser[6] observed an average of 7 revisions per group in a single, large project, but provided no overhead figure. In his paper on DSEE[5], Leblang reported that delta storage combined with blank compression results in an overhead of a mere 1–2 per cent per revision. Since leading blanks accounted for about 20 per cent of the surveyed Pascal programs, a revision group with 5–10 members was smaller than a single cleartext copy.

The above observations demonstrate clearly that the space needed for extra revisions is small. With delta storage, the luxury of keeping multiple revisions online is certainly affordable. In fact, introducing a system with delta storage may reduce storage requirements, because programmers often save back-up copies anyway. Since back-up copies are stored much more efficiently with deltas, introducing a system such as RCS may actually free a considerable amount of space.

7. Survey of Version Control Tools

The need to keep back-up copies of software arose when programs and data were no longer stored on paper media, but were entered from terminals and stored on disk. Back-up copies are desirable for reliability, and many modern editors automatically save a back-up copy for every file touched. This strategy is valuable for short-term back-ups, but not suitable for long-term version control, since an existing back-up copy is overwritten whenever the corresponding file is edited.

Tape archives are suitable for long-term, offline storage. If all changed files are dumped on a back-up tape once per day, old revisions remain accessible. However, tape archives are unsatisfactory for version control in several ways. First, backing up the file system every 24 hours does not capture intermediate revisions. Secondly, the old revisions are not online, and accessing them is tedious and time-consuming. In particular, it is impractical to compare several old revisions of a group, because that

may require mounting and searching several tapes. Tape archives are important fail-safe tools in the event of catastrophic disk failures or accidental deletions, but they are ill-suited for version control. Conversely, version control tools do not obviate the need for tape archives.

A natural technique for keeping several old revisions online is to never delete a file. Editing a file simply creates a new file with the same name, but with a different sequence number. This technique, available as an option in DEC's VMS operating system, turns out to be inadequate for version control. First, it is prohibitively expensive in terms of storage costs, especially since no data compression techniques are employed. Secondly, indiscriminately storing every change produces too many revisions, and programmers have difficulties distinguishing them. The proliferation of revisions forces programmers to spend much time on finding and deleting useless files. Thirdly, most of the support functions like locking, logging, revision selection, and identification described in this paper are not available.

An alternative approach is to separate editing from revision control. The user may repeatedly edit a given revision, until freezing it with an explicit command. Once a revision is frozen, it is stored permanently and can no longer be modified. (In RCS, freezing a revisions is done with *ci*.) Editing a frozen revision implicitly creates a new one, which can again be changed repeatedly until it is frozen itself. This approach saves exactly those revisions that the user considers important, and keeps the number of revisions manageable. IBM's CLEAR/CASTER[7], AT&T's SCCS[3], CMU's SDC[8] and DEC's CMS[9], are examples of version control systems using this approach. CLEAR/CASTER maintains a data base of programs, specifications, documentation and messages, using deltas. Its goal is to provide control over the development process from a management viewpoint. SCCS stores multiple revisions of source text in an ancestral tree, records a log entry for each revision, provides access control, and has facilities for uniquely identifying each revision. An efficient delta technique reduces the space consumed by each revision group. SDC is much simpler than SCCS because it stores not more than two revisions. However, it maintains a complete log for all old revisions, some of which may be on back-up tape. CMS, like SCCS, manages tree-structured revision groups, but offers no identification mechanism.

Tools for dealing with configurations are still in a state of flux. SCCS, SDC and CMS can be combined with MAKE or MAKE-like programs. Since flexible selection rules are missing from all these tools, it is sometimes difficult to specify precisely which revision of each group should be passed to MAKE for building a desired configuration. The Xerox Cedar system[10] provides a 'System Modeller' that can rebuild a configuration from an arbitrary set of module revisions. The revisions of a module are only distinguished by creation time, and there is no tool for managing groups. Since the selection rules are primitive, the System Modeller appears to be somewhat tedious to use. Apollo's DSEE[5] is a sophisticated software engineering environment. It manages revision groups in a way similar to SCCS and CMS. Configurations are built using 'configuration threads'. A configuration thread states which revision of each group named in a configuration should be chosen. A configuration thread may contain dynamic specifiers (e.g., 'choose the revisions I am currently working on, and the most recent revisions otherwise'), which are bound automatically at build time. It also provides a notification mechanism for alerting maintainers about the need to rebuild a system after a change.

RCS is based on a general model for describing multi-version/multi-configuration systems[11]. The model describes systems using AND/OR graphs, where AND nodes represent configurations, and OR nodes represent version groups. The model gives rise to a suit of selection rules for composing configurations, almost all of which are implemented in RCS. The revisions selected by RCS are passed to MAKE for configuration building. Revision group management is modelled after SCCS. RCS retains SCCS's best features, but offers a significantly simpler user interface, flexible selection rules, adequate integration with MAKE and improved identification. A detailed comparison of RCS and SCCS appears in Reference 4.

An important component of all revision control systems is a program for computing deltas. SCCS and RCS use the program *diff*[2], which first computes the longest common substring of two revisions, and then produces the delta from that substring. The delta is simply an edit script consisting of deletion and insertion commands that generate one revision from the other.

A delta based on a longest common substring is not necessarily minimal, because it does not take advantage of crossing block moves. Crossing block moves arise if two or more blocks of lines (e.g., procedures) appear in a different order in two revisions. An edit script derived from a longest common

substring first deletes the shorter of the two blocks, and then reinserts it. Heckel[12] proposed an algorithm for detecting block moves, but since the algorithm is based on heuristics, there are conditions under which the generated delta is far from minimal. DSEE uses this algorithm combined with blank compression, apparently with satisfactory overall results. A new algorithm that is guaranteed to produce a minimal delta based on block moves appears in Reference 13. A future release of RCS will use this algorithm.

Acknowledgements: Many people have helped make RCS a success by contributed criticisms, suggestions, corrections, and even whole new commands (including manual pages). The list of people is too long to be reproduced here, but my sincere thanks for their help and goodwill goes to all of them.

Appendix: Synopsis of RCS Operations

ci **– check in revisions**

Ci stores the contents of a working file into the corresponding RCS file as a new revision. If the RCS file doesn't exist, *ci* creates it. *Ci* removes the working file, unless one of the options *–u* or *–l* is present. For each check-in, *ci* asks for a commentary describing the changes relative to the previous revision.

Ci assigns the revision number given by the *–r* option; if that option is missing, it derives the number from the lock held by the user; if there is no lock and locking is not strict, *ci* increments the number of the latest revision on the trunk. A side branch can only be started by explicitly specifying its number with the *–r* option during check-in.

Ci also determines whether the revision to be checked in is different from the previous one, and asks whether to proceed if not. This facility simplifies check-in operations for large systems, because one need not remember which files were changed.

The option *–k* searches the checked in file for identification markers containing the attributes revision number, check-in date, author and state, and assigns these to the new revision rather than computing them. This option is useful for software distribution: Recipients of distributed software using RCS should check in updates with the *–k* option. This convention guarantees that revision numbers, check-in dates, etc., are the same at all sites.

co **– check out revisions**

Co retrieves revisions according to revision number, date, author and state attributes. It either places the revision into the working file, or prints it on the standard output. *Co* always expands the identification markers.

ident **– extract identification markers**

Ident extracts the identification markers expanded by *co* from any file and prints them.

rcs **– change RCS file attributes**

Rcs is an administrative operation that changes access lists, locks, unlocks, breaks locks, toggles the strict-locking feature, sets state attributes and symbolic revision numbers, changes the description, and deletes revisions. A revision can only be deleted if it is not the fork of a side branch.

rcsclean **– clean working directory**

Rcsclean removes working files that were checked out but never changed.*

rcsdiff **– compare revisions**

Rcsdiff compares two revisions and prints their difference, using the UNIX tool *diff*. One of the revisions compared may be checked out. This command is useful for finding out about changes.

rcsfreeze **– freeze a configuration**

Rcsfreeze assigns the same symbolic revision number to a given revision in all RCS files. This command is useful for accurately recording a configuration.*

rcsmerge **– merge revisions**

Rcsmerge merges two revisions, *rev1* and *rev2*, with respect to a common ancestor. A 3-way file comparison determines the segments of lines that are (a) the same in all three revisions, or (b) the same in 2 revisions, or (c) different in all three. For all segments of type (b) where *rev1* is the differing revision, the segment in *rev1* replaces the corresponding segment of *rev2*. Type (c) indicates an overlapping change, is flagged as an error, and requires user intervention to select the correct alternative.

rlog **– read log messages**

Rlog prints the log messages and other information in an RCS file.

---

* The *rcsclean* and *rcsfreeze* commands are optional and are not always installed.

References

1. Feldman, Stuart I., "Make—A Program for Maintaining Computer Programs," *Software—Practice & Experience*, vol. 9, no. 3, pp. 255-265, March 1979.
2. Hunt, James W. and McIlroy, M. D., "An Algorithm for Differential File Comparison," 41, Computing Science Technical Report, Bell Laboratories, June 1976.
3. Rochkind, Marc J., "The Source Code Control System," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 4, pp. 364-370, Dec. 1975.
4. Tichy, Walter F., "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, pp. 58-67, ACM, IEEE, IPS, NBS, September 1982.
5. Leblang, David B. and Chase, Robert P., "Computer-Aided Software Engineering in a Distributed Workstation Environment," *SIGPLAN Notices*, vol. 19, no. 5, pp. 104-112, May 1984. Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.
6. Glasser, Alan L., "The Evolution of a Source Code Control System," *Software Engineering Notes*, vol. 3, no. 5, pp. 122-125, Nov. 1978. Proceedings of the Software Quality and Assurance Workshop.
7. Brown, H.B., "The Clear/Caster System," *Nato Conference on Software Engineering, Rome*, 1970.
8. Habermann, A. Nico, *A Software Development Control System,* Technical Report, Carnegie-Mellon University, Department of Computer Science, Jan. 1979.
9. DEC, *Code Management System,* Digital Equipment Corporation, 1982. Document No. EA-23134-82
10. Lampson, Butler W. and Schmidt, Eric E., "Practical Use of a Polymorphic Applicative Language," in *Proceedings of the 10th Symposium on Principles of Programming Languages*, pp. 237-255, ACM, January 1983.
11. Tichy, Walter F., "A Data Model for Programming Support Environments and its Application," in *Automated Tools for Information System Design and Development*, ed. by Hans-Jochen Schneider and Anthony I. Wasserman, North-Holland Publishing Company, Amsterdam, 1982.
12. Heckel, Paul, "A Technique for Isolating Differences Between Files," *Communications of the ACM*, vol. 21, no. 4, pp. 264-268, April 1978.
13. Tichy, Walter F., "The String-to-String Correction Problem with Block Moves," *ACM Transactions on Computer Systems*, vol. 2, no. 4, pp. 309-321, Nov. 1984.

An Introduction to the Source Code Control System

Eric Allman
Project Ingres
University of California at Berkeley

This document gives a quick introduction to using the Source Code Control System (SCCS). The presentation is geared to programmers who are more concerned with what to do to get a task done rather than how it works; for this reason some of the examples are not well explained. For details of what the magic options do, see the section on ''Further Information''.

> This is a working document. Please send any comments or suggestions to eric@Berkeley.Edu.

1. Introduction

SCCS is a source management system. Such a system maintains a record of versions of a system; a record is kept with each set of changes of what the changes are, why they were made, and who made them and when. Old versions can be recovered, and different versions can be maintained simultaneously. In projects with more than one person, SCCS will insure that two people are not editing the same file at the same time.

All versions of your program, plus the log and other information, is kept in a file called the ''s-file''. There are three major operations that can be performed on the s-file:

(1) Get a file for compilation (not for editing). This operation retrieves a version of the file from the s-file. By default, the latest version is retrieved. This file is intended for compilation, printing, or whatever; it is specifically NOT intended to be edited or changed in any way; any changes made to a file retrieved in this way will probably be lost.

(2) Get a file for editing. This operation also retrieves a version of the file from the s-file, but this file is intended to be edited and then incorporated back into the s-file. Only one person may be editing a file at one time.

(3) Merge a file back into the s-file. This is the companion operation to (2). A new version number is assigned, and comments are saved explaining why this change was made.

2. Learning the Lingo

There are a number of terms that are worth learning before we go any farther.

2.1. S-file

The s-file is a single file that holds all the different versions of your file. The s-file is stored in differential format; *i.e.*, only the differences between versions are stored, rather than the entire text of the new version. This saves disk space and allows selective changes to be removed later. Also included in the s-file is some header information for each version, including the comments given by the person

---

This is version 1.21 of this document. It was last modified on 12/5/80.

who created the version explaining why the changes were made.

2.2. Deltas

Each set of changes to the s-file (which is approximately [but not exactly!] equivalent to a version of the file) is called a *delta*. Although technically a delta only includes the *changes* made, in practice it is usual for each delta to be made with respect to all the deltas that have occurred before[1]. However, it is possible to get a version of the file that has selected deltas removed out of the middle of the list of changes – equivalent to removing your changes later.

2.3. SID's (or, version numbers)

A SID (SCCS Id) is a number that represents a delta. This is normally a two-part number consisting of a "release" number and a "level" number. Normally the release number stays the same, however, it is possible to move into a new release if some major change is being made.

Since all past deltas are normally applied, the SID of the final delta applied can be used to represent a version number of the file as a whole.

2.4. Id keywords

When you get a version of a file with intent to compile and install it (*i.e.*, something other than edit it), some special keywords are expanded inline by SCCS. These *Id Keywords* can be used to include the current version number or other information into the file. All id keywords are of the form **%***x***%**, where *x* is an upper case letter. For example, **%I%** is the SID of the latest delta applied, **%W%** includes the module name, SID, and a mark that makes it findable by a program, and **%G%** is the date of the latest delta applied. There are many others, most of which are of dubious usefulness.

When you get a file for editing, the id keywords are not expanded; this is so that after you put them back in to the s-file, they will be expanded automatically on each new version. But notice: if you were to get them expanded accidently, then your file would appear to be the same version forever more, which would of course defeat the purpose. Also, if you should install a version of the program without expanding the id keywords, it will be impossible to tell what version it is (since all it will have is "%W%" or whatever).

3. Creating SCCS Files

To put source files into SCCS format, run the following shell script from csh:

```
mkdir SCCS save
foreach i (*.[ch])
        sccs admin –i$i $i
        mv $i save/$i
end
```

This will put the named files into s-files in the subdirectory "SCCS" The files will be removed from the current directory and hidden away in the directory "save", so the next thing you will probably want to do is to get all the files (described below). When you are convinced that SCCS has correctly created the s-files, you should remove the directory "save".

If you want to have id keywords in the files, it is best to put them in before you create the s-files. If you do not, *admin* will print "No Id Keywords (cm7)", which is a warning message only.

---

[1]This matches normal usage, where the previous changes are not saved at all, so all changes are automatically based on all other changes that have happened through history.

4. Getting Files for Compilation

To get a copy of the latest version of a file, run

```
sccs get prog.c
```

SCCS will respond:

```
1.1
87 lines
```

meaning that version 1.1 was retrieved[2] and that it has 87 lines. The file *prog.c* will be created in the current directory. The file will be read-only to remind you that you are not supposed to change it.

This copy of the file should not be changed, since SCCS is unable to merge the changes back into the s-file. If you do make changes, they will be lost the next time someone does a *get*.

5. Changing Files (or, Creating Deltas)

5.1. Getting a copy to edit

To edit a source file, you must first get it, requesting permission to edit it[3]:

```
sccs edit prog.c
```

The response will be the same as with *get* except that it will also say:

```
New delta 1.2
```

You then edit it, using a standard text editor:

```
vi prog.c
```

5.2. Merging the changes back into the s-file

When the desired changes are made, you can put your changes into the SCCS file using the *delta* command:

```
sccs delta prog.c
```

Delta will prompt you for "comments?" before it merges the changes in. At this prompt you should type a one-line description of what the changes mean (more lines can be entered by ending each line except the last with a backslash[4]). *Delta* will then type:

```
1.2
5 inserted
3 deleted
84 unchanged
```

saying that delta 1.2 was created, and it inserted five lines, removed three lines, and left 84 lines unchanged[5]. The *prog.c* file will be removed; it can be retrieved using *get*.

---

[2]Actually, the SID of the final delta applied was 1.1.

[3]The "edit" command is equivalent to using the –e flag to *get*, as:

```
sccs get –e prog.c
```

Keep this in mind when reading other documentation.

[4]Yes, this is a stupid default.

[5]Changes to a line are counted as a line deleted and a line inserted.

5.3. When to make deltas

It is probably unwise to make a delta before every recompilation or test; otherwise, you tend to get a lot of deltas with comments like "fixed compilation problem in previous delta" or "fixed botch in 1.3". However, it is very important to delta everything before installing a module for general use. A good technique is to edit the files you need, make all necessary changes and tests, compiling and editing as often as necessary without making deltas. When you are satisfied that you have a working version, delta everything being edited, re-get them, and recompile everything.

5.4. What's going on: the info command

To find out what files where being edited, you can use:

```
sccs info
```

to print out all the files being edited and other information such as the name of the user who did the edit. Also, the command:

```
sccs check
```

is nearly equivalent to the *info* command, except that it is silent if nothing is being edited, and returns non-zero exit status if anything is being edited; it can be used in an "install" entry in a makefile to abort the install if anything has not been properly deltaed.

If you know that everything being edited should be deltaed, you can use:

```
sccs delta `sccs tell`
```

The *tell* command is similar to *info* except that only the names of files being edited are output, one per line.

All of these commands take a **–b** flag to ignore "branches" (alternate versions, described later) and the **–u** flag to only give files being edited by you. The **–u** flag takes an optional *user* argument, giving only files being edited by that user. For example,

```
sccs info –ujohn
```

gives a listing of files being edited by john.

5.5. ID keywords

Id keywords can be inserted into your file that will be expanded automatically by *get*. For example, a line such as:

```
static char SccsId[] = "%W%\t%G%";
```

will be replaced with something like:

```
static char SccsId[] = "@(#)prog.c	1.2	08/29/80";
```

This tells you the name and version of the source file and the time the delta was created. The string "@(#)" is a special string which signals the beginning of an SCCS Id keyword.

5.5.1. The what command

To find out what version of a program is being run, use:

```
sccs what prog.c /usr/bin/prog
```

which will print all strings it finds that begin with "@(#)". This works on all types of files, including binaries and libraries. For example, the above command will output something like:

```
prog.c:
        prog.c  1.2     08/29/80
/usr/bin/prog:
        prog.c  1.1     02/05/79
```

From this I can see that the source that I have in prog.c will not compile into the same version as the binary in /usr/bin/prog.

5.5.2. Where to put id keywords

ID keywords can be inserted anywhere, including in comments, but Id Keywords that are compiled into the object module are especially useful, since it lets you find out what version of the object is being run, as well as the source. However, there is a cost: data space is used up to store the keywords, and on small address space machines this may be prohibitive.

When you put id keywords into header files, it is important that you assign them to different variables. For example, you might use:

```
static char AccessSid[] = "%W% %G%";
```

in the file *access.h* and:

```
static char OpsysSid[] = "%W% %G%";
```

in the file *opsys.h*. Otherwise, you will get compilation errors because ‘‘SccsId’’ is redefined. The problem with this is that if the header file is included by many modules that are loaded together, the version number of that header file is included in the object module many times; you may find it more to your taste to put id keywords in header files in comments.

5.6. Keeping SID's consistent across files

With some care, it is possible to keep the SID's consistent in multi-file systems. The trick here is to always *edit* all files at once. The changes can then be made to whatever files are necessary and then all files (even those not changed) are redeltaed. This can be done fairly easily by just specifying the name of the directory that the SCCS files are in:

```
sccs edit SCCS
```

which will *edit* all files in that directory. To make the delta, use:

```
sccs delta SCCS
```

You will be prompted for comments only once.

5.7. Creating new releases

When you want to create a new release of a program, you can specify the release number you want to create on the *edit* command. For example:

```
sccs edit –r2 prog.c
```

will cause the next delta to be in release two (that is, it will be numbered 2.1). Future deltas will automatically be in release two. To change the release number of an entire system, use:

```
sccs edit –r2 SCCS
```

6. Restoring Old Versions

6.1. Reverting to old versions

Suppose that after delta 1.2 was stable you made and released a delta 1.3. But this introduced a bug, so you made a delta 1.4 to correct it. But 1.4 was still buggy, and you decided you wanted to go back to the old version. You could revert to delta 1.2 by choosing the SID in a get:

```
sccs get –r1.2 prog.c
```

This will produce a version of *prog.c* that is delta 1.2 that can be reinstalled so that work can proceed.

In some cases you don't know what the SID of the delta you want is. However, you can revert to the version of the program that was running as of a certain date by using the **–c** (cutoff) flag. For

example,

```
sccs get –c800722120000 prog.c
```

will retrieve whatever version was current as of July 22, 1980 at 12:00 noon. Trailing components can be stripped off (defaulting to their highest legal value), and punctuation can be inserted in the obvious places; for example, the above line could be equivalently stated:

```
sccs get –c"80/07/22 12:00:00" prog.c
```

6.2. Selectively deleting old deltas

Suppose that you later decided that you liked the changes in delta 1.4, but that delta 1.3 should be removed. You could do this by *excluding* delta 1.3:

```
sccs edit –x1.3 prog.c
```

When delta 1.5 is made, it will include the changes made in delta 1.4, but will exclude the changes made in delta 1.3. You can exclude a range of deltas using a dash. For example, if you want to get rid of 1.3 and 1.4 you can use:

```
sccs edit –x1.3–1.4 prog.c
```

which will exclude all deltas from 1.3 to 1.4. Alternatively,

```
sccs edit –x1.3–1 prog.c
```

will exclude a range of deltas from 1.3 to the current highest delta in release 1.

In certain cases when using **–x** (or **–i**; see below) there will be conflicts between versions; for example, it may be necessary to both include and delete a particular line. If this happens, SCCS always prints out a message telling the range of lines effected; these lines should then be examined very carefully to see if the version SCCS got is ok.

Since each delta (in the sense of ‘‘a set of changes’’) can be excluded at will, that this makes it most useful to put each semantically distinct change into its own delta.

7. Auditing Changes

7.1. The prt command

When you created a delta, you presumably gave a reason for the delta to the ‘‘comments?’’ prompt. To print out these comments later, use:

```
sccs prt prog.c
```

This will produce a report for each delta of the SID, time and date of creation, user who created the delta, number of lines inserted, deleted, and unchanged, and the comments associated with the delta. For example, the output of the above command might be:

```
D 1.2    80/08/29 12:35:31    bill    2    1    00005/00003/00084
removed "-q" option

D 1.1    79/02/05 00:19:31    eric    1    0    00087/00000/00000
date and time created 80/06/10 00:19:31 by eric
```

7.2. Finding why lines were inserted

To find out why you inserted lines, you can get a copy of the file with each line preceded by the SID that created it:

```
sccs get –m prog.c
```

You can then find out what this delta did by printing the comments using *prt*.

To find out what lines are associated with a particular delta (*e.g.*, 1.3), use:

```
sccs get –m –p prog.c | grep ´^1.3´
```

The **–p** flag causes SCCS to output the generated source to the standard output rather than to a file.

7.3. Finding what changes you have made

When you are editing a file, you can find out what changes you have made using:

```
sccs diffs prog.c
```

Most of the ‘‘diff’’ flags can be used. To pass the **–c** flag, use **–C**.

To compare two versions that are in deltas, use:

```
sccs sccsdiff -r1.3 -r1.6 prog.c
```

to see the differences between delta 1.3 and delta 1.6.

8. Shorthand Notations

There are several sequences of commands that get executed frequently. *Sccs* tries to make it easy to do these.

8.1. Delget

A frequent requirement is to make a delta of some file and then get that file. This can be done by using:

```
sccs delget prog.c
```

which is entirely equivalent to using:

```
sccs delta prog.c
sccs get prog.c
```

The ‘‘deledit’’ command is equivalent to ‘‘delget’’ except that the ‘‘edit’’ command is used instead of the ‘‘get’’ command.

8.2. Fix

Frequently, there are small bugs in deltas, e.g., compilation errors, for which there is no reason to maintain an audit trail. To *replace* a delta, use:

```
sccs fix –r1.4 prog.c
```

This will get a copy of delta 1.4 of prog.c for you to edit and then delete delta 1.4 from the SCCS file. When you do a delta of prog.c, it will be delta 1.4 again. The –r flag must be specified, and the delta that is specified must be a leaf delta, i.e., no other deltas may have been made subsequent to the creation of that delta.

8.3. Unedit

If you found you edited a file that you did not want to edit, you can back out by using:

```
sccs unedit prog.c
```

8.4. The –d flag

If you are working on a project where the SCCS code is in a directory somewhere, you may be able to simplify things by using a shell alias. For example, the alias:

```
alias syssccs sccs –d/usr/src
```

will allow you to issue commands such as:

```
sysscs edit cmd/who.c
```

which will look for the file ‘‘/usr/src/cmd/SCCS/who.c’’. The file ‘‘who.c’’ will always be created in your current directory regardless of the value of the –d flag.

9. Using SCCS on a Project

Working on a project with several people has its own set of special problems. The main problem occurs when two people modify a file at the same time. SCCS prevents this by locking an s-file while it is being edited.

As a result, files should not be reserved for editing unless they are actually being edited at the time, since this will prevent other people on the project from making necessary changes. For example, a good scenario for working might be:

```
sccs edit a.c g.c t.c
vi a.c g.c t.c
# do testing of the (experimental) version
sccs delget a.c g.c t.c
sccs info
# should respond "Nothing being edited"
make install
```

As a general rule, all source files should be deltaed before installing the program for general use. This will insure that it is possible to restore any version in use at any time.

10. Saving Yourself

10.1. Recovering a munged edit file

Sometimes you may find that you have destroyed or trashed a file that you were trying to edit[6]. Unfortunately, you can’t just remove it and re-*edit* it; SCCS keeps track of the fact that someone is trying to edit it, so it won’t let you do it again. Neither can you just get it using *get*, since that would expand the Id keywords. Instead, you can say:

```
sccs get –k prog.c
```

This will not expand the Id keywords, so it is safe to do a delta with it.

Alternately, you can *unedit* and *edit* the file.

10.2. Restoring the s-file

In particularly bad circumstances, the SCCS file itself may get munged. The most common way this happens is that it gets edited. Since SCCS keeps a checksum, you will get errors every time you read the file. To fix this checksum, use:

```
sccs admin –z prog.c
```

11. Using the Admin Command

There are a number of parameters that can be set using the *admin* command. The most interesting of these are flags. Flags can be added by using the **–f** flag. For example:

```
sccs admin –fd1 prog.c
```

sets the ‘‘d’’ flag to the value ‘‘1’’. This flag can be deleted by using:

---

[6]Or given up and decided to start over.

```
sccs admin –dd prog.c
```

The most useful flags are:

b Allow branches to be made using the –b flag to *edit*.

d*SID* Default SID to be used on a *get* or *edit*. If this is just a release number it constrains the version to a particular release only.

i Give a fatal error if there are no Id Keywords in a file. This is useful to guarantee that a version of the file does not get merged into the s-file that has the Id Keywords inserted as constants instead of internal forms.

y The ''type'' of the module. Actually, the value of this flag is unused by SCCS except that it replaces the **%Y%** keyword.

The **–t***file* flag can be used to store descriptive text from *file*. This descriptive text might be the documentation or a design and implementation document. Using the **–t** flag insures that if the SCCS file is sent, the documentation will be sent also. If *file* is omitted, the descriptive text is deleted. To see the descriptive text, use ''prt –t''.

The *admin* command can be used safely any number of times on files. A file need not be gotten for *admin* to work.

12. Maintaining Different Versions (Branches)

Sometimes it is convenient to maintain an experimental version of a program for an extended period while normal maintenance continues on the version in production. This can be done using a ''branch.'' Normally deltas continue in a straight line, each depending on the delta before. Creating a branch ''forks off'' a version of the program.

The ability to create branches must be enabled in advance using:

```
sccs admin –fb prog.c
```

The **–fb** flag can be specified when the SCCS file is first created.

12.1. Creating a branch

To create a branch, use:

```
sccs edit –b prog.c
```

This will create a branch with (for example) SID 1.5.1.1. The deltas for this version will be numbered 1.5.1.*n*.

12.2. Getting from a branch

Deltas in a branch are normally not included when you do a get. To get these versions, you will have to say:

```
sccs get –r1.5.1 prog.c
```

12.3. Merging a branch back into the main trunk

At some point you will have finished the experiment, and if it was successful you will want to incorporate it into the release version. But in the meantime someone may have created a delta 1.6 that you don't want to lose. The commands:

```
sccs edit –i1.5.1.1–1.5.1 prog.c
sccs delta prog.c
```

will merge all of your changes into the release system. If some of the changes conflict, get will print an error; the generated result should be carefully examined before the delta is made.

12.4. A more detailed example

The following technique might be used to maintain a different version of a program. First, create a directory to contain the new version:

```
mkdir ../newxyz
cd ../newxyz
```

Edit a copy of the program on a branch:

```
sccs –d../xyz edit prog.c
```

When using the old version, be sure to use the **–b** flag to info, check, tell, and clean to avoid confusion. For example, use:

```
sccs info –b
```

when in the directory ‘‘xyz’’.

If you want to save a copy of the program (still on the branch) back in the s-file, you can use:

```
sccs -d../xyz deledit prog.c
```

which will do a delta on the branch and reedit it for you.

When the experiment is complete, merge it back into the s-file using delta:

```
sccs -d../xyz delta prog.c
```

At this point you must decide whether this version should be merged back into the trunk (*i.e.* the default version), which may have undergone changes. If so, it can be merged using the **–i** flag to *edit* as described above.

12.5. A warning

Branches should be kept to a minimum. After the first branch from the trunk, SID’s are assigned rather haphazardly, and the structure gets complex fast.

13. Using SCCS with Make

SCCS and make can be made to work together with a little care. A few sample makefiles for common applications are shown.

There are a few basic entries that every makefile ought to have. These are:

a.out	(or whatever the makefile generates.) This entry regenerates whatever this makefile is supposed to regenerate. If the makefile regenerates many things, this should be called ‘‘all’’ and should in turn have dependencies on everything the makefile can generate.
install	Moves the objects to the final resting place, doing any special *chmod*’s or *ranlib*’s as appropriate.
sources	Creates all the source files from SCCS files.
clean	Removes all files from the current directory that can be regenerated from SCCS files.
print	Prints the contents of the directory.

The examples shown below are only partial examples, and may omit some of these entries when they are deemed to be obvious.

The *clean* entry should not remove files that can be regenerated from the SCCS files. It is sufficiently important to have the source files around at all times that the only time they should be removed is when the directory is being mothballed. To do this, the command:

```
sccs clean
```

can be used. This will remove all files for which an s-file exists, but which is not being edited.

13.1. To maintain single programs

Frequently there are directories with several largely unrelated programs (such as simple commands). These can be put into a single makefile:

```
LDFLAGS= –i –s

prog: prog.o
        $(CC) $(LDFLAGS) –o prog prog.o
prog.o: prog.c prog.h

example: example.o
        $(CC) $(LDFLAGS) –o example example.o
example.o: example.c

.DEFAULT:
        sccs get $<
```

The trick here is that the .DEFAULT rule is called every time something is needed that does not exist, and no other rule exists to make it. The explicit dependency of the **.o** file on the **.c** file is important. Another way of doing the same thing is:

```
SRCS= prog.c prog.h example.c

LDFLAGS= –i –s

prog: prog.o
        $(CC) $(LDFLAGS) –o prog prog.o
prog.o: prog.h

example: example.o
        $(CC) $(LDFLAGS) –o example example.o

sources: $(SRCS)
$(SRCS):
        sccs get $@
```

There are a couple of advantages to this approach: (1) the explicit dependencies of the .o on the .c files are not needed, (2) there is an entry called "sources" so if you want to get all the sources you can just say ‘‘make sources’’, and (3) the makefile is less likely to do confusing things since it won’t try to *get* things that do not exist.

13.2. To maintain a library

Libraries that are largely static are best updated using explicit commands, since *make* doesn’t know about updating them properly. However, libraries that are in the process of being developed can be handled quite adequately. The problem is that the .o files have to be kept out of the library as well as in the library.

```
# configuration information
OBJS=  a.o b.o c.o d.o
SRCS=  a.c b.c c.c d.s x.h y.h z.h
TARG= /usr/lib

# programs
GET=   sccs get
REL=
AR=    –ar
RANLIB=        ranlib

lib.a: $(OBJS)
        $(AR) rvu lib.a $(OBJS)
        $(RANLIB) lib.a

install: lib.a
        sccs check
        cp lib.a $(TARG)/lib.a
        $(RANLIB) $(TARG)/lib.a

sources: $(SRCS)
$(SRCS):
        $(GET) $(REL) $@

print: sources
        pr *.h *.[cs]
clean:
        rm –f *.o
        rm –f core a.out $(LIB)
```

The "$(REL)" in the get can be used to get old versions easily; for example:

```
make b.o REL=–r1.3
```

The *install* entry includes the line "sccs check" before anything else. This guarantees that all the s-files are up to date (*i.e.*, nothing is being edited), and will abort the *make* if this condition is not met.

13.3. To maintain a large program

```
OBJS=  a.o b.o c.o d.o
SRCS=  a.c b.c c.y d.s x.h y.h z.h

GET=   sccs get
REL=

a.out: $(OBJS)
        $(CC) $(LDFLAGS) $(OBJS) $(LIBS)

sources: $(SRCS)
$(SRCS):
        $(GET) $(REL) $@
```

(The *print* and *clean* entries are identical to the previous case.) This makefile requires copies of the source and object files to be kept during development. It is probably also wise to include lines of the form:

```
a.o: x.h y.h
b.o: z.h
c.o: x.h y.h z.h
z.h: x.h
```

so that modules will be recompiled if header files change.

Since *make* does not do transitive closure on dependencies, you may find in some makefiles lines like:

```
z.h: x.h
        touch z.h
```

This would be used in cases where file z.h has a line:

```
#include "x.h"
```

in order to bring the mod date of z.h in line with the mod date of x.h. When you have a makefile such as above, the *touch* command can be removed completely; the equivalent effect will be achieved by doing an automatic *get* on z.h.

14. Further Information

The *SCCS/PWB User's Manual* gives a deeper description of how to use SCCS. Of particular interest are the numbering of branches, the l-file, which gives a description of what deltas were used on a get, and certain other SCCS commands.

The SCCS manual pages are a good last resort. These should be read by software managers and by people who want to know everything about everything.

Both of these documents were written without the *sccs* front end in mind, so most of the examples are slightly different from those in this document.

Quick Reference

1. Commands

The following commands should all be preceded with ‘‘sccs’’. This list is not exhaustive; for more options see *Further Information*.

get — Gets files for compilation (not for editing). Id keywords are expanded.

- –r*SID* Version to get.
- –p Send to standard output rather than to the actual file.
- –k Don’t expand id keywords.
- –i*list* List of deltas to include.
- –x*list* List of deltas to exclude.
- –m Precede each line with SID of creating delta.
- –c*date* Don’t apply any deltas created after *date.*

edit — Gets files for editing. Id keywords are not expanded. Should be matched with a *delta* command.

- –r*SID* Same as *get*. If *SID* specifies a release that does not yet exist, the highest numbered delta is retrieved and the new delta is numbered with *SID*.
- –b Create a branch.
- –i*list* Same as *get*.
- –x*list* Same as *get*.

delta — Merge a file gotten using *edit* back into the s-file. Collect comments about why this delta was made.

unedit — Remove a file that has been edited previously without merging the changes into the s-file.

prt — Produce a report of changes.

- –t Print the descriptive text.
- –e Print (nearly) everything.

info — Give a list of all files being edited.

- –b Ignore branches.
- –u[*user*] Ignore files not being edited by *user*.

check — Same as *info*, except that nothing is printed if nothing is being edited and exit status is returned.

tell — Same as *info*, except that one line is produced per file being edited containing only the file name.

clean — Remove all files that can be regenerated from the s-file.

what — Find and print id keywords.

admin — Create or set parameters on s-files.

- –i*file* Create, using *file* as the initial contents.
- –z Rebuild the checksum in case the file has been trashed.

–f*flag* Turn on the *flag*.

–d*flag* Turn off (delete) the *flag*.

–t*file* Replace the descriptive text in the s-file with the contents of *file*. If *file* is omitted, the text is deleted. Useful for storing documentation or ‘‘design & implementation’’ documents to insure they get distributed with the s-file.

Useful flags are:

b Allow branches to be made using the –b flag to *edit.*

d*SID* Default SID to be used on a *get* or *edit*.

i Cause ‘‘No Id Keywords’’ error message to be a fatal error rather than a warning.

t The module ‘‘type’’; the value of this flag replaces the **%Y%** keyword.

fix Remove a delta and reedit it.

delget Do a *delta* followed by a *get*.

deledit Do a *delta* followed by an *edit*.

2. Id Keywords

%Z% Expands to ‘‘@(#)’’ for the *what* command to find.

%M% The current module name, *e.g.,* ‘‘prog.c’’.

%I% The highest SID applied.

%W% A shorthand for ‘‘%Z%%M% <tab> %I%’’.

%G% The date of the delta corresponding to the ‘‘%I%’’ keyword.

%R% The current release number, *i.e.*, the first component of the ‘‘%I%’’ keyword.

%Y% Replaced by the value of the **t** flag (set by *admin*).

Yacc: Yet Another Compiler-Compiler

Stephen C. Johnson

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Computer program input generally has some structure; in fact, every computer program that does input can be thought of as defining an "input language" which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often are lax about checking their inputs for validity.

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.

Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

In addition to compilers for C, APL, Pascal, RATFOR, etc., Yacc has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a Fortran debugging system.

0: Introduction

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a *parser*, calls the user-supplied low-level input routine (the *lexical analyzer*) to pick up the basic items (called *tokens*) from the input stream. These tokens are organized according to the input structure rules, called *grammar rules* ; when one of these rules has been recognized, then user code supplied for this rule, an *action*, is invoked; actions have the ability to return values and make use of the values of other actions.

Yacc is written in a portable dialect of C[1] and the actions, and output subroutine, are in C as well. Moreover, many of the syntactic conventions of Yacc follow C.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

```
date : month_name day ´,´ year ;
```

Here, *date*, *month_name*, *day*, and *year* represent structures of interest in the input process; presumably, *month_name*, *day*, and *year* are defined elsewhere. The comma "," is enclosed in single quotes; this implies

that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

```
July 4, 1776
```

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a *terminal symbol*, while the structure recognized by the parser is called a *nonterminal symbol*. To avoid confusion, terminal symbols will usually be referred to as *tokens*.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

```
month_name : 'J' 'a' 'n' ;
month_name : 'F' 'e' 'b' ;

    . . .

month_name : 'D' 'e' 'c' ;
```

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and *month_name* would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond Yacc's ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a *month_name* was seen; in this case, *month_name* would be a token.

Literal characters such as "," must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is realively easy to add to the above example the rule

```
date : month '/' day '/' year ;
```

allowing

```
7 / 4 / 1776
```

as a synonym for

```
July 4, 1776
```

In most cases, this new rule could be "slipped in" to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, Yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self contradictory, or they may require a more powerful recognition mechanism than that available to Yacc. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While Yacc cannot handle all possible specifications, its power compares favorably with similar systems; moreover, the constructions which are difficult for Yacc to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid Yacc specifications for their input revealed errors of conception or design early in the program development.

The theory underlying Yacc has been described elsewhere.[2][3][4] Yacc has been extensively used in numerous practical applications, including *lint*,[5] the Portable C Compiler,[6] and a system for typesetting mathematics.[7]

The next several sections describe the basic process of preparing a Yacc specification; Section 1 describes the preparation of grammar rules, Section 2 the preparation of the user supplied actions associated with these rules, and Section 3 the preparation of lexical analyzers. Section 4 describes the operation of the parser. Section 5 discusses various reasons why Yacc may be unable to produce a parser from a specification, and what to do about it. Section 6 describes a simple mechanism for handling operator precedences in arithmetic expressions. Section 7 discusses error detection and recovery. Section 8 discusses the operating environment and special features of the parsers Yacc produces. Section 9 gives some suggestions which should improve the style and efficiency of the specifications. Section 10 discusses some advanced topics, and Section 11 gives acknowledgements. Appendix A has a brief example, and Appendix B gives a summary of the Yacc input syntax. Appendix C gives an example using some of the more advanced features of Yacc, and, finally, Appendix D describes mechanisms and syntax no longer actively supported, but provided for historical continuity with older versions of Yacc.

1: Basic Specifications

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. In addition, for reasons discussed in Section 3, it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections: the *declarations*, *(grammar) rules*, and *programs*. The sections are separated by double percent "%%" marks. (The percent "%" is generally used in Yacc specifications as an escape character.)

In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal Yacc specification is

```
%%
rules
```

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* . . . */, as in C and PL/I.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A : BODY ;
```

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are Yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot ".", underscore "_", and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes "'". As in C, the backslash "\" is an escape character within literals, and all the C escapes are recognized. Thus

'\n' newline
'\r' return
'\'' single quote "'"
'\\' backslash "\"
'\t' tab
'\b' backspace
'\f' form feed
'\xxx' "xxx" in octal

For a number of technical reasons, the NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, the vertical bar "|" can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

```
A    :    B  C  D  ;
A    :    E  F  ;
A    :    G  ;
```

can be given to Yacc as

```
A    :    B  C  D
     |    E  F
     |    G
     ;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

```
empty :  ;
```

Names representing tokens must be declared; this is most simply done by writing

```
%token  name1  name2 . . .
```

in the declarations section. (See Sections 3 , 5, and 6 for much more discussion). Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the *start symbol*, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the %start keyword:

```
%start  symbol
```

The end of the input to the parser is signaled by a special token, called the *endmarker*. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the endmarker is seen; it *accepts* the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate; see section 3, below. Usually the endmarker represents some reasonably obvious I/O status, such as "end-of-file" or "end-of-record".

2: Actions

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions.

Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces "{" and "}". For example,

```
A     :     '(' B ')'
                {     hello( 1, "abc" );  }
```

and

```
XXX :     YYY  ZZZ
                {     printf("a message\n");
                      flag = 25;    }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol "dollar sign" "$" is used as a signal to Yacc in this context.

To return a value, the action normally sets the pseudo-variable "$$" to some value. For example, an action that does nothing but return the value 1 is

```
{  $$ = 1;  }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, . . ., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

```
A     :     B  C  D  ;
```

for example, then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, consider the rule

```
expr  :     '(' expr ')'  ;
```

The value returned by this rule is usually the value of the *expr* in parentheses. This can be indicated by

```
expr  :     '(' expr ')'          {  $$ = $2 ;  }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form

```
A     :     B     ;
```

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. Yacc permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

```
A     :     B
                {  $$ = 1;  }
            C
                {   x = $2;   y = $3;  }
      ;
```

the effect is to set *x* to 1, and *y* to the value returned by C.

Actions that do not terminate a rule are actually handled by Yacc by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. Yacc actually treats the above example as if it had been written:

```
$ACT:     /* empty */
                { $$ = 1; }
    ;

A   :     B $ACT C
                { x = $2; y = $3; }
    ;
```

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function *node*, written so that the call

```
node( L, n1, n2 )
```

creates a node with label L, and descendants n1 and n2, and returns the index of the newly created node. Then parse tree can be built by supplying actions such as:

```
expr :    expr '+' expr
                { $$ = node( '+', $1, $3 ); }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks "%{" and "%}". These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{  int variable = 0;  %}
```

could be placed in the declarations section, making *variable* accessible to all of the actions. The Yacc parser uses only names beginning in "yy"; the user should avoid such names.

In these examples, all the values are integers: a discussion of values of other types will be found in Section 10.

3: Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called *yylex*. The function returns an integer, the *token number*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable *yylval*.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by Yacc, or chosen by the user. In either case, the "# define" mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the Yacc specification file. The relevant portion of the lexical analyzer might look like:

```
yylex(){
        extern int yylval;
        int c;
        . . .
        c = getchar();
        . . .
        switch( c ) {
                . . .
        case ´0´:
        case ´1´:
          . . .
        case ´9´:
                yylval = c–´0´;
                return( DIGIT );
                . . .
                }
        . . .
```

The intent is to return a token number of DIGIT, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT will be defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of token names *if* or *while* will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name *error* is reserved for error handling, and should not be used naively (see Section 7).

As mentioned above, the token numbers may be chosen by Yacc or by the user. In the default situation, the numbers are chosen by Yacc. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

To assign a token number to a token (including literals), the first appearance of the token name or literal *in the declarations section* can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user; thus, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the *Lex* program developed by Mike Lesk.[8] These lexical analyzers are designed to work in close harmony with Yacc parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. Lex can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) which do not fit any theoretical framework, and whose lexical analyzers must be crafted by hand.

4: How the Parser Works

Yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the *lookahead* token). The *current state* is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called *shift*, *reduce*, *accept*, and *error*. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls *yylex* to obtain the next token.
2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

The *shift* action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

```
IF      shift 34
```

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

The *reduce* action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a ".") is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

```
.       reduce 18
```

refers to *grammar rule* 18, while the action

```
IF      shift 34
```

refers to *state* 34.

Suppose the rule being reduced is

```
A   :   x y z   ;
```

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped equals the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack while recognizing x, y, and z, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left hand symbol and an ordinary shift of a token, however, so this action is called a *goto* action. In particular, the lookahead token is cleared by a shift, and is not affected by a goto. In any case, the uncovered state contains an entry such as:

```
A       goto 20
```

causing state 20 to be pushed onto the stack, and become the current state.

In effect, the reduce action "turns back the clock" in the parse, popping the states off the stack to go back to the state where the right hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right hand side of the rule is empty, no states are popped off of the stack: the uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable *yylval* is copied onto the value stack. After the return from the user code, the reduction is carried out. When the *goto* action is done, the external variable *yyval* is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The *accept* action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The *error* action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The parser reports an error, and attempts to recover the situation and resume parsing: the error recovery (as opposed to the detection of error) will be covered in Section 7.

It is time for an example! Consider the specification

```
%token DING DONG DELL
%%
rhyme      :      sound place
      ;
sound:     DING DONG
      ;
place :    DELL
      ;
```

When Yacc is invoked with the **–v** option, a file called *y.output* is produced, with a human-readable description of the parser. The *y.output* file corresponding to the above grammar (with some statistics stripped off the end) is:

```
state 0
        $accept : _rhyme $end

        DING shift 3
        . error

        rhyme goto 1
        sound goto 2

state 1
        $accept : rhyme_$end

        $end accept
        . error

state 2
        rhyme : sound_place

        DELL shift 5
        . error

        place goto 4

state 3
        sound : DING_DONG

        DONG shift 6
        . error

state 4
        rhyme : sound place_ (1)

        . reduce 1

state 5
        place : DELL_ (3)

        . reduce 3

state 6
        sound : DING DONG_ (2)

        . reduce 2
```

Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The _ character is used to indicate what has been seen, and what is yet to come, in each rule. Suppose the input is

```
DING DONG DELL
```

It is instructive to follow the steps of the parser while processing this input.

Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, *DING*, is read, becoming the lookahead token. The action in state 0 on *DING* is is ‘‘shift 3’’, so state 3 is pushed onto the stack, and the lookahead token is cleared. State 3 becomes the current state. The next token, *DONG*, is read, becoming the lookahead token. The action in state 3

on the token *DONG* is "shift 6", so state 6 is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the lookahead, the parser reduces by rule 2.

```
sound : DING DONG
```

This rule has two symbols on the right hand side, so two states, 6 and 3, are popped off of the stack, uncovering state 0. Consulting the description of state 0, looking for a goto on *sound*,

```
sound goto 2
```

is obtained; thus state 2 is pushed onto the stack, becoming the current state.

In state 2, the next token, *DELL*, must be read. The action is "shift 5", so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the lookahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on *place*, the left side of rule 3, is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on *rhyme* causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained, indicated by "$end" in the *y.output* file. The action in state 1 when the endmarker is seen is to accept, successfully ending the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as *DING DONG DONG*, *DING DONG*, *DING DONG DELL DELL*, etc. A few minutes spend with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

5: Ambiguity and Conflicts

A set of grammar rules is *ambiguous* if there is some input string that can be structured in two or more different ways. For example, the grammar rule

```
expr : expr '-' expr
```

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

```
expr - expr - expr
```

the rule allows this input to be structured as either

```
( expr - expr ) - expr
```

or as

```
expr - ( expr - expr )
```

(The first is called *left association*, the second *right association*).

Yacc detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given an input such as

```
expr - expr - expr
```

When the parser has read the second expr, the input that it has seen:

```
expr - expr
```

matches the right side of the grammar rule above. The parser could *reduce* the input by applying this rule; after applying the rule; the input is reduced to *expr* (the left side of the rule). The parser would then read the final part of the input:

```
- expr
```

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, when the parser has seen

```
expr – expr
```

it could defer the immediate application of the rule, and continue reading the input until it had seen

```
expr – expr – expr
```

It could then apply the rule to the rightmost three symbols, reducing them to *expr* and leaving

```
expr – expr
```

Now the rule can be reduced once more; the effect is to take the right associative interpretation. Thus, having read

```
expr – expr
```

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a *shift / reduce conflict*. It may also happen that the parser has a choice of two legal reductions; this is called a *reduce / reduce conflict*. Note that there are never any ''Shift/shift'' conflicts.

When there are shift/reduce or reduce/reduce conflicts, Yacc still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a *disambiguating rule*.

Yacc invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.
2. In a reduce/reduce conflict, the default is to reduce by the *earlier* grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided whenever possible.

Conflicts may arise because of mistakes in input or logic, or because the grammar rules, while consistent, require a more complex parser than Yacc can construct. The use of actions within rules can also cause conflicts, if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate, and leads to an incorrect parser. For this reason, Yacc always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural, and produces slower parsers; thus, Yacc will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an ''if-then-else'' construction:

```
stat  :     IF ´(´ cond ´)´ stat
      |     IF ´(´ cond ´)´ stat ELSE stat
      ;
```

In these rules, *IF* and *ELSE* are tokens, *cond* is a nonterminal symbol describing conditional (logical) expressions, and *stat* is a nonterminal symbol describing statements. The first rule will be called the *simple-if* rule, and the second the *if-else* rule.

These two rules form an ambiguous construction, since input of the form

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be structured according to these rules in two ways:

```
IF ( C1 ) {
    IF ( C2 ) S1
    }
ELSE S2
```

or

```
IF ( C1 ) {
    IF ( C2 ) S1
    ELSE S2
    }
```

The second interpretation is the one given in most programming languages having this construct. Each *ELSE* is associated with the last preceding "un-*ELSE*'d" *IF*. In this example, consider the situation where the parser has seen

```
IF ( C1 ) IF ( C2 ) S1
```

and is looking at the *ELSE*. It can immediately reduce by the simple-if rule to get

```
IF ( C1 ) stat
```

and then read the remaining input,

```
ELSE S2
```

and reduce

```
IF ( C1 ) stat ELSE S2
```

by the if-else rule. This leads to the first of the above groupings of the input.

On the other hand, the *ELSE* may be shifted, *S2* read, and then the right hand portion of

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be reduced by the if-else rule to get

```
IF ( C1 ) stat
```

which can be reduced by the simple-if rule. This leads to the second of the above groupings of the input, which is usually desired.

Once again the parser can do two valid things – there is a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This shift/reduce conflict arises only when there is a particular current input symbol, *ELSE*, and particular inputs already seen, such as

```
IF ( C1 ) IF ( C2 ) S1
```

In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of Yacc are best understood by examining the verbose (**–v**) option output file. For example, the output corresponding to the above conflict state might be:

```
23: shift/reduce conflict (shift 45, reduce 18) on ELSE

state 23

        stat : IF ( cond ) stat_          (18)
        stat : IF ( cond ) stat_ELSE stat

      ELSE     shift 45
      .        reduce 18
```

The first line describes the conflict, giving the state and the input symbol. The ordinary state description follows, giving the grammar rules active in the state, and the parser actions. Recall that the underline marks the portion of the grammar rules which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

```
IF ( cond ) stat
```

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is *ELSE*, it is possible to shift into state 45. State 45 will have, as part of its description, the line

```
stat : IF ( cond ) stat ELSE_stat
```

since the *ELSE* will have been shifted in this state. Back in state 23, the alternative action, described by "**.**", is to be done if the input symbol is not mentioned explicitly in the above actions; thus, in this case, if the input symbol is not *ELSE*, the parser reduces by grammar rule 18:

```
stat : IF ´(´ cond ´)´ stat
```

Once again, notice that the numbers following "shift" commands refer to other states, while the numbers following "reduce" commands refer to grammar rule numbers. In the *y.output* file, the rule numbers are printed after those rules which can be reduced. In most one states, there will be at most reduce action possible in the state, and this will be the default command. The user who encounters unexpected shift/reduce conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate. In really tough cases, the user might need to know more about the behavior and construction of the parser than can be covered here. In this case, one of the theoretical references[2,3,4] might be consulted; the services of a local guru might also be appropriate.

6: Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of *precedence* levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
expr : expr OP expr
```

and

```
expr : UNARY expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the user specifies the precedence, or binding strength, of all the operators, and the associativity of the binary operators. This information is sufficient to allow Yacc to resolve the parsing conflicts in accordance with these rules, and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a Yacc keyword: %left, %right, or %nonassoc, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed

in order of increasing precedence or binding strength. Thus,

```
%left ´+´ ´–´
%left ´*´ ´/´
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star and slash, which are also left associative. The keyword %right is used to describe right associative operators, and the keyword %nonassoc is used to describe operators, like the operator .LT. in Fortran, that may not associate with themselves; thus,

```
A .LT. B .LT. C
```

is illegal in Fortran, and such an operator would be described with the keyword %nonassoc in Yacc. As an example of the behavior of these declarations, the description

```
%right ´=´
%left ´+´ ´–´
%left ´*´ ´/´

%%

expr : expr ´=´ expr
     | expr ´+´ expr
     | expr ´–´ expr
     | expr ´*´ expr
     | expr ´/´ expr
     | NAME
     ;
```

might be used to structure the input

```
a = b = c*d – e – f*g
```

as follows:

```
a = ( b = ( ((c*d)–e) – (f*g) ) )
```

When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation, but different precedences. An example is unary and binary ´–´; unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, %prec, changes the precedence level associated with a particular grammar rule. %prec appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, to make unary minus have the same precedence as multiplication the rules might resemble:

```
%left ´+´ ´–´
%left ´*´ ´/´

%%

expr : expr ´+´ expr
     | expr ´–´ expr
     | expr ´*´ expr
     | expr ´/´ expr
     | ´–´ expr %prec ´*´
     | NAME
     ;
```

A token declared by %left, %right, and %nonassoc need not be, but may be, declared by %token as well.

The precedences and associativities are used by Yacc to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals that have them.
2. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the %prec construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.
3. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.
4. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by Yacc. This means that mistakes in the specification of precedences may disguise errors in the input grammar; it is a good idea to be sparing with precedences, and use them in an essentially ''cookbook'' fashion, until some experience has been gained. The *y.output* file is very useful in deciding whether the parser is actually doing what was intended.

7: Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found; it is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser ''restarted'' after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, Yacc provides a simple, but reasonably general, feature. The token name ''error'' is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. The parser pops its stack until it enters a state where the token ''error'' is legal. It then behaves as if the token ''error'' were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form

```
stat    :    error
```

would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement, and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general, but difficult to control. Somewhat easier are rules such as

```
stat    :    error  ´;´
```

Here, when there is an error, the parser attempts to skip over the statement, but will do so by skipping to the

next ´;´. All tokens after the error and before the next ´;´ cannot be shifted, and are discarded. When the ´;´ is seen, this rule will be reduced, and any ‘‘cleanup’’ action associated with it performed.

Another form of error rule arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error rule might be

```
input :     error  ^\n´  {  printf( "Reenter last line: " );  }  input
                   {     $$  =  $4;  }
```

There is one potential difficulty with this approach; the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example is better written

```
input :     error  ^\n´
                   {     yyerrok;
                         printf( "Reenter last line: " );   }
            input
                   {     $$  =  $4;  }
        ;
```

As mentioned above, the token seen immediately after the ‘‘error’’ symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after error were to call some sophisticated resynchronization routine, supplied by the user, that attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by yylex would presumably be the first token in a legal statement; the old, illegal token must be discarded, and the error state reset. This could be done by a rule like

```
stat    :   error
                   {     resynch();
                         yyerrok ;
                         yyclearin ;   }
        ;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors; moreover, the user can get control to deal with the error actions required by other portions of the program.

8: The Yacc Environment

When the user inputs a specification to Yacc, the output is a file of C programs, called *y.tab.c* on most systems (due to local file system conventions, the names may differ from installation to installation). The function produced by Yacc is called *yyparse* ; it is an integer valued function. When it is called, it in turn repeatedly calls *yylex* , the lexical analyzer supplied by the user (see Section 3) to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) *yyparse* returns the value 1, or the lexical analyzer returns the endmarker token and the parser accepts. In this case, *yyparse* returns the value 0.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C program, a program called *main* must be defined, that eventually calls *yyparse* . In addition, a routine called *yyerror* prints a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using Yacc, a library has been provided with default versions of *main* and *yyerror*. The name of this library is system dependent; on many systems the library is accessed by a **–ly** argument to the loader. To show the triviality of these default programs, the source is given below:

```
main(){
        return( yyparse() );
        }
```

and

```
# include <stdio.h>

yyerror(s) char *s; {
        fprintf( stderr, "%s\n", s );
        }
```

The argument to *yyerror* is a string containing an error message, usually the string ''syntax error''. The average application will want to do better than this. Ordinarily, the program should keep track of the input line number, and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the lookahead token number at the time the error was detected; this may be of some interest in giving better diagnostics. Since the *main* program is probably supplied by the user (to read arguments, etc.) the Yacc library is useful only in small projects, or in the earliest stages of larger ones.

The external integer variable *yydebug* is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

9: Hints for Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following style hints owe much to Brian Kernighan.

a. Use all capital letters for token names, all lower case letters for nonterminal names. This rule comes under the heading of ''knowing who to blame when things go wrong.''

b. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.

c. Put all rules with the same left hand side together. Put the left hand side in only once, and let all following rules begin with a vertical bar.

d. Put a semicolon only after the last rule with a given left hand side, and put the semicolon on a separate line. This allows new rules to be easily added.

e. Indent rule bodies by two tab stops, and action bodies by three tab stops.

The example in Appendix A is written following this style, as are the examples in the text of this paper (where space permits). The user must make up his own mind about these stylistic questions; the central problem, however, is to make the rules visible through the morass of action code.

Left Recursion

The algorithm used by the Yacc parser encourages so called ''left recursive'' grammar rules: rules of the form

```
name :      name  rest_of_rule  ;
```

These rules frequently arise when writing specifications of sequences and lists:

```
list    :       item
        |       list ´,´ item
        ;
```

and

```
seq     :       item
        |       seq  item
        ;
```

In each of these cases, the first rule will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as

```
seq     :       item
        |       item  seq
        ;
```

the parser would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion wherever reasonable.

It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
seq     :       /* empty */
        |       seq  item
        ;
```

Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if Yacc is asked to decide which empty sequence it has seen, when it hasn't seen enough to know!

Lexical Tie-ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names might be entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of 0 or more declarations, followed by 0 or more statements. Consider:

```
%{
        int dflag;
%}
  ... other declarations ...

%%

prog  :       decls  stats
      ;

decls :       /* empty */
                      {       dflag = 1;  }
      |       decls  declaration
      ;

stats :       /* empty */
                      {       dflag = 0;  }
      |       stats  statement
      ;

    ... other rules ...
```

The flag *dflag* is now 0 when reading statements, and 1 when reading declarations, *except for the first token in the first statement.* This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of ‘‘backdoor’’ approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Reserved Words

Some programming languages permit the user to use words like ‘‘if’’, which are normally reserved, as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of Yacc; it is difficult to pass information to the lexical analyzer telling it ‘‘this instance of ‘if’ is a keyword, and that instance is a variable’’. The user can make a stab at it, using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be *reserved* ; that is, be forbidden for use as variable names. There are powerful stylistic reasons for preferring this, anyway.

10: Advanced Topics

This section discusses a number of advanced features of Yacc.

Simulating Error and Accept in Actions

The parsing actions of error and accept can be simulated in an action by use of macros YYACCEPT and YYERROR. YYACCEPT causes *yyparse* to return the value 0; YYERROR causes the parser to behave as if the current input symbol had been a syntax error; *yyerror* is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple endmarkers or context-sensitive syntax checking.

Accessing Values in Enclosing Rules.

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit, but in this case the digit may be 0 or negative. Consider

```
sent  :     adj  noun  verb  adj  noun
                  {  look at the sentence . . .  }
      ;

adj   :     THE         {     $$ = THE;  }
      |     YOUNG       {     $$ = YOUNG;  }
      . . .
      ;

noun  :     DOG
                  {     $$ = DOG;  }
      |     CRONE
                  {     if( $0 == YOUNG ){
                              printf( "what?\n" );
                              }
                        $$ = CRONE;
                        }
      ;
      . . .
```

In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG. Obviously, this is only possible when a great deal is known about what might precede the symbol *noun* in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism will save a great deal of trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. Yacc can also support values of other types, including structures. In addition, Yacc keeps track of the types, and inserts appropriate union member names so that the resulting parser will be strictly type checked. The Yacc value stack (see Section 4) is declared to be a *union* of the various types of values desired. The user declares the union, and associates union member names to each token and nonterminal symbol having a value. When the value is referenced through a $$ or $n construction, Yacc will automatically insert the appropriate union name, so that no unwanted conversions will take place. In addition, type checking commands such as *Lint*[5] will be far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union; this must be done by the user since other programs, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where Yacc can not easily determine the type.

To declare the union, the user includes in the declaration section:

```
%union  {
        body of union ...
        }
```

This declares the Yacc value stack, and the external variables *yylval* and *yyval*, to have type equal to this union. If Yacc was invoked with the **–d** option, the union declaration is copied onto the *y.tab.h* file. Alternatively, the union may be declared in a header file, and a typedef used to define the variable YYSTYPE to represent this union. Thus, the header file might also have said:

```
typedef union {
        body of union ...
        } YYSTYPE;
```

The header file must be included in the declarations section, by use of %{ and %}.

Once YYSTYPE is defined, the union member names must be associated with the various terminal and nonterminal names. The construction

< name >

is used to indicate a union member name. If this follows one of the keywords %token, %left, %right, and %nonassoc, the union member name is associated with the tokens listed. Thus, saying

%left <optype> '+' '-'

will cause any reference to values returned by these two tokens to be tagged with the union member name *optype*. Another keyword, %type, is used similarly to associate union member names with nonterminals. Thus, one might say

%type <nodetype> expr stat

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no *a priori* type. Similarly, reference to left context values (such as $0 – see the previous subsection) leaves Yacc with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name, between < and >, immediately after the first $. An example of this usage is

```
rule :  aaa { $<intval>$ = 3; } bbb
            { fun( $<intval>2, $<other>0 ); }
     ;
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in Appendix C. The facilities in this subsection are not triggered until they are used: in particular, the use of %type will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of $n or $$ to refer to something with no defined type is diagnosed. If these facilities are not triggered, the Yacc value stack is used to hold *int*'s, as was true historically.

11: Acknowledgements

Yacc owes much to a most stimulating collection of users, who have goaded me beyond my inclination, and frequently beyond my ability, in their endless search for ''one more feature''. Their irritating unwillingness to learn how to do things my way has usually led to my doing things their way; most of the time, they have been right. B. W. Kernighan, P. J. Plauger, S. I. Feldman, C. Imagna, M. E. Lesk, and A. Snyder will recognize some of their ideas in the current version of Yacc. C. B. Haley contributed to the error recovery algorithm. D. M. Ritchie, B. W. Kernighan, and M. O. Harris helped translate this document into English. Al Aho also deserves special credit for bringing the mountain to Mohammed, and other favors.

References

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
2. A. V. Aho and S. C. Johnson, "LR Parsing," *Comp. Surveys*, vol. 6, no. 2, pp. 99-124, June 1974.
3. A. V. Aho, S. C. Johnson, and J. D. Ullman, "Deterministic Parsing of Ambiguous Grammars," *Comm. Assoc. Comp. Mach.*, vol. 18, no. 8, pp. 441-452, August 1975.
4. A. V. Aho and J. D. Ullman, *Principles of Compiler Design,* Addison-Wesley, Reading, Mass., 1977.
5. S. C. Johnson, "Lint, a C Program Checker," Comp. Sci. Tech. Rep. No. 65, 1978. updated version TM 78-1273-3
6. S. C. Johnson, "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages*, pp. 97-104, January 1978.
7. B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," *Comm. Assoc. Comp. Mach.*, vol. 18, pp. 151-157, Bell Laboratories, Murray Hill, New Jersey, March 1975.
8. M. E. Lesk, "Lex — A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey, October 1975.

Appendix A: A Simple Example

This example gives the complete Yacc specification for a small desk calculator; the desk calculator has 26 registers, labeled ''a'' through ''z'', and accepts arithmetic expressions made up of the operators +, –, ∗, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression at the top level is an assignment, the value is not printed; otherwise it is. As in C, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a Yacc specification, the desk calculator does a reasonable job of showing how precedences and ambiguities are used, and demonstrating simple error recovery. The major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately, line by line. Note the way that decimal and octal integers are read in by the grammar rules; This job is probably better done by the lexical analyzer.

```
%{
# include <stdio.h>
# include <ctype.h>

int regs[26];
int base;

%}

%start list

%token DIGIT LETTER

%left ´|´
%left ´&´
%left ´+´ ´–´
%left ´∗´ ´/´ ´%´
%left UMINUS     /∗ supplies precedence for unary minus ∗/

%%     /∗ beginning of rules section ∗/

list    :       /∗ empty ∗/
        |       list stat ´\n´
        |       list error ´\n´
                        {       yyerrok; }
        ;

stat    :       expr
                        {       printf( "%d\n", $1 ); }
        |       LETTER ´=´ expr
                        {       regs[$1] = $3; }
        ;

expr    :       ´(´ expr ´)´
                        {       $$ = $2; }
        |       expr ´+´ expr
                        {       $$ = $1 + $3; }
        |       expr ´–´ expr
                        {       $$ = $1 – $3; }
        |       expr ´∗´ expr
                        {       $$ = $1 ∗ $3; }
```

```
    |   expr '/' expr
                        {       $$ = $1 / $3; }
        |       expr '%' expr
                        {       $$ = $1 % $3; }
        |       expr '&' expr
                        {       $$ = $1 & $3; }
        |       expr '|' expr
                        {       $$ = $1 | $3; }
        |       '-' expr        %prec UMINUS
                        {       $$ = - $2; }
        |       LETTER
                        {       $$ = regs[$1]; }
        |       number
        ;

number :        DIGIT
                        {       $$ = $1;    base = ($1==0) ? 8 : 10; }
        |       number DIGIT
                        {       $$ = base * $1 + $2; }
        ;

%%      /* start of programs */

yylex() {               /* lexical analysis routine */
          /* returns LETTER for a lower case letter, yylval = 0 through 25 */
          /* return DIGIT for a digit, yylval = 0 through 9 */
          /* all other characters are returned immediately */

        int c;

        while( (c=getchar()) == ' ' ) { /* skip blanks */ }

        /* c is now nonblank */

        if( islower( c ) ) {
                yylval = c - 'a';
                return ( LETTER );
                }
        if( isdigit( c ) ) {
                yylval = c - '0';
                return( DIGIT );
                }
        return( c );
        }
```

Appendix B: Yacc Input Syntax

This Appendix has a description of the Yacc input syntax, as a Yacc specification. Context dependencies, etc., are not considered. Ironically, the Yacc input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule, immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier, and decide whether the next token (skipping blanks, newlines, comments, etc.) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERS, but never as part of C_IDENTIFIERs.

```
        /* grammar for the input to Yacc */

        /* basic entities */
%token  IDENTIFIER      /* includes identifiers and literals */
%token  C_IDENTIFIER    /* identifier (but not literal) followed by colon */
%token  NUMBER          /* [0-9]+ */

        /* reserved words:  %type => TYPE, %left => LEFT, etc. */

%token  LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token  MARK    /* the %% mark */
%token  LCURL   /* the %{ mark */
%token  RCURL   /* the %} mark */

        /* ascii character literals stand for themselves */

%start  spec

%%

spec    :       defs MARK rules tail
        ;

tail    :       MARK { In this action, eat up the rest of the file }
        |       /* empty: the second MARK is optional */
        ;

defs    :       /* empty */
        |       defs def
        ;

def     :       START IDENTIFIER
        |       UNION { Copy union definition to output }
        |       LCURL { Copy C code to output file } RCURL
        |       ndefs rword tag nlist
        ;
```

```
rword   :       TOKEN
        |       LEFT
        |       RIGHT
        |       NONASSOC
        |       TYPE
        ;

tag     :       /* empty: union tag is optional */
        |       '<' IDENTIFIER '>'
        ;

nlist   :       nmno
        |       nlist nmno
        |       nlist ',' nmno
        ;

nmno    :       IDENTIFIER              /* NOTE: literal illegal with %type */
        |       IDENTIFIER NUMBER       /* NOTE: illegal with %type */
        ;

        /* rules section */

rules   :       C_IDENTIFIER rbody prec
        |       rules rule
        ;

rule    :       C_IDENTIFIER rbody prec
        |       '|' rbody prec
        ;

rbody   :       /* empty */
        |       rbody IDENTIFIER
        |       rbody act
        ;

act     :       '{' { Copy action, translate $$, etc. } '}'
        ;

prec    :       /* empty */
        |       PREC IDENTIFIER
        |       PREC IDENTIFIER act
        |       prec ';'
        ;
```

Appendix C: An Advanced Example

This Appendix gives an example of a grammar using some of the advanced features discussed in Section 10. The desk calculator example in Appendix A is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants, the arithmetic operations +, –, ∗, /, unary –, and = (assignment), and has 26 floating point variables, "a" through "z". Moreover, it also understands *intervals*, written

(x , y)

where x is less than or equal to y. There are 26 interval valued variables "A" through "Z" that may also be used. The usage is similar to that in Appendix A; assignments return no value, and print nothing, while expressions print the (floating or interval) value.

This example explores a number of interesting features of Yacc and C. Intervals are represented by a structure, consisting of the left and right endpoint values, stored as *double*'s. This structure is given a type name, INTERVAL, by using *typedef*. The Yacc value stack can also contain floating point scalars, and integers (used to index into the arrays holding the variable values). Notice that this entire strategy depends strongly on being able to assign structures and unions in C. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of YYERROR to handle error conditions: division by an interval containing 0, and an interval presented in the wrong order. In effect, the error recovery mechanism of Yacc is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (e.g. scalar or interval) of intermediate expressions. Note that a scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through Yacc: 18 Shift/Reduce and 26 Reduce/Reduce. The problem can be seen by looking at the two input lines:

2.5 + (3.5 – 4.)

and

2.5 + (3.5 , 4.)

Notice that the 2.5 is to be used in an interval valued expression in the second example, but this fact is not known until the "," is read; by this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator: one when the left operand is a scalar, and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflicts will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive, but not very general. If there were many kinds of expression types, instead of just two, the number of rules needed would increase dramatically, and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value, and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C library routine *atof* is used to do the actual conversion from a character string to a double precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar, provoking a syntax error in the parser, and thence error recovery.

```
%{

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

```
typedef struct interval {
	double lo, hi;
	} INTERVAL;

INTERVAL vmul(), vdiv();

double atof();

double dreg[ 26 ];
INTERVAL vreg[ 26 ];

%}

%start	lines

%union	{
	int ival;
	double dval;
	INTERVAL vval;
	}

%token <ival> DREG VREG		/* indices into dreg, vreg arrays */

%token <dval> CONST		/* floating point constant */

%type <dval> dexp		/* expression */

%type <vval> vexp		/* interval expression */

	/* precedence information about the operators */

%left '+' '-'
%left '*' '/'
%left UMINUS		/* precedence for unary minus */

%%

lines :	/* empty */
	|	lines line
	;

line :	dexp '\n'
			{	printf( "%15.8f\n", $1 ); }
	|	vexp '\n'
			{	printf( "(%15.8f , %15.8f )\n", $1.lo, $1.hi ); }
	|	DREG '=' dexp '\n'
			{	dreg[$1] = $3; }
	|	VREG '=' vexp '\n'
			{	vreg[$1] = $3; }
	|	error '\n'
			{	yyerrok; }
	;

dexp :	CONST
```

```
    |   DREG
                {       $$ = dreg[$1];  }
        |   dexp '+' dexp
                {       $$ = $1 + $3;   }
        |   dexp '-' dexp
                {       $$ = $1 - $3;   }
        |   dexp '*' dexp
                {       $$ = $1 * $3;   }
        |   dexp '/' dexp
                {       $$ = $1 / $3;   }
        |   '-' dexp    %prec UMINUS
                {       $$ = -$2;   }
        |   '(' dexp ')'
                {       $$ = $2;    }
        ;

vexp :  dexp
                {       $$.hi = $$.lo = $1;     }
        |   '(' dexp ',' dexp ')'
                {
                $$.lo = $2;
                $$.hi = $4;
                if( $$.lo > $$.hi ){
                        printf( "interval out of order\n" );
                        YYERROR;
                        }
                }
        |   VREG
                {       $$ = vreg[$1];      }
        |   vexp '+' vexp
                {       $$.hi = $1.hi + $3.hi;
                        $$.lo = $1.lo + $3.lo;      }
        |   dexp '+' vexp
                {       $$.hi = $1 + $3.hi;
                        $$.lo = $1 + $3.lo;     }
        |   vexp '-' vexp
                {       $$.hi = $1.hi - $3.lo;
                        $$.lo = $1.lo - $3.hi;      }
        |   dexp '-' vexp
                {       $$.hi = $1 - $3.lo;
                        $$.lo = $1 - $3.hi;     }
        |   vexp '*' vexp
                {       $$ = vmul( $1.lo, $1.hi, $3 );  }
        |   dexp '*' vexp
                {       $$ = vmul( $1, $1, $3 );    }
        |   vexp '/' vexp
                {       if( dcheck( $3 ) ) YYERROR;
                        $$ = vdiv( $1.lo, $1.hi, $3 );  }
        |   dexp '/' vexp
                {       if( dcheck( $3 ) ) YYERROR;
                        $$ = vdiv( $1, $1, $3 );    }
        |   '-' vexp    %prec UMINUS
                {       $$.hi = -$2.lo;     $$.lo = -$2.hi;     }
        |   '(' vexp ')'
```

```
            {       $$ = $2; }
        ;

%%

# define BSZ 50         /* buffer size for floating point numbers */

        /* lexical analysis */

yylex(){
        register c;

        while( (c=getchar()) == ´ ´ ){ /* skip over blanks */ }

        if( isupper( c ) ){
                yylval.ival = c – ´A´;
                return( VREG );
                }
        if( islower( c ) ){
                yylval.ival = c – ´a´;
                return( DREG );
                }

        if( isdigit( c ) || c==´.´ ){
                /* gobble up digits, points, exponents */

                char buf[BSZ+1], *cp = buf;
                int dot = 0, exp = 0;

                for( ; (cp–buf)<BSZ ; ++cp,c=getchar() ){

                        *cp = c;
                        if( isdigit( c ) ) continue;
                        if( c == ´.´ ){
                                if( dot++ || exp ) return( ´.´ );   /* will cause syntax error */
                                continue;
                                }

                        if( c == ´e´ ){
                                if( exp++ ) return( ´e´ );    /* will cause syntax error */
                                continue;
                                }

                        /* end of number */
                        break;
                        }
                *cp = ´\0´;
                if( (cp–buf) >= BSZ ) printf( "constant too long: truncated\n" );
                else ungetc( c, stdin );   /* push back last char read */
                yylval.dval = atof( buf );
                return( CONST );
                }
        return( c );
        }
```

```
INTERVAL hilo( a, b, c, d ) double a, b, c, d; {
	/* returns the smallest interval containing a, b, c, and d */
	/* used by *, / routines */
	INTERVAL v;

	if( a>b ) { v.hi = a;  v.lo = b; }
	else { v.hi = b;  v.lo = a; }

	if( c>d ) {
		if( c>v.hi ) v.hi = c;
		if( d<v.lo ) v.lo = d;
		}
	else {
		if( d>v.hi ) v.hi = d;
		if( c<v.lo ) v.lo = c;
		}
	return( v );
	}

INTERVAL vmul( a, b, v ) double a, b;  INTERVAL v; {
	return( hilo( a*v.hi, a*v.lo, b*v.hi, b*v.lo ) );
	}

dcheck( v ) INTERVAL v; {
	if( v.hi >= 0. && v.lo <= 0. ){
		printf( "divisor interval contains 0.\n" );
		return( 1 );
		}
	return( 0 );
	}

INTERVAL vdiv( a, b, v ) double a, b;  INTERVAL v; {
	return( hilo( a/v.hi, a/v.lo, b/v.hi, b/v.lo ) );
	}
```

Appendix D: Old Features Supported but not Encouraged

This Appendix mentions synonyms and features which are supported for historical continuity, but, for various reasons, are not encouraged.

1. Literals may also be delimited by double quotes ‘‘"’’.

2. Literals may be more than one character long. If all the characters are alphabetic, numeric, or _, the type number of the literal is defined, just as if the literal did not have the quotes around it. Otherwise, it is difficult to find the value for such literals.

   The use of multi-character literals is likely to mislead those unfamiliar with Yacc, since it suggests that Yacc is doing a job which must be actually done by the lexical analyzer.

3. Most places where % is legal, backslash ‘‘\’’ may be used. In particular, \\ is the same as %%, \left the same as %left, etc.

4. There are a number of other synonyms:

   %< is the same as %left
   %> is the same as %right
   %binary and %2 are the same as %nonassoc
   %0 and %term are the same as %token
   %= is the same as %prec

5. Actions may also have the form

   ={ . . . }

   and the curly braces can be dropped if the action is a single C statement.

6. C code between %{ and %} used to be permitted at the head of the rules section, as well as in the declaration section.

Lex – A Lexical Analyzer Generator

M. E. Lesk and E. Schmidt

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it.

Lex can generate analyzers in either C or Ratfor, a language which can be translated automatically to portable Fortran. It is available on the PDP-11 UNIX, Honeywell GCOS, and IBM OS systems. This manual, however, will only discuss generating analyzers in C on the UNIX system, which is the only supported form of Lex under UNIX Version 7. Lex is designed to simplify interfacing with Yacc, for those with access to this compiler-compiler system.

1. Introduction.

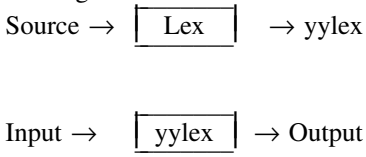
Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.

Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called "host languages." Just as general purpose

languages can produce code to run on different computer hardware, Lex can write code in different host languages. The host language is used for the output code generated by Lex and also for the program fragments added by the user. Compatible run-time libraries for the different host languages are also provided. This makes Lex adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. At present, the only supported host language is C, although Fortran (in the form of Ratfor [2] has been available in the past. Lex itself exists on UNIX, GCOS, and OS/370; but the code generated by Lex may be taken anywhere the appropriate compilers exist.

Lex turns the user's expressions and actions (called *source* in this memo) into the host general-purpose language; the generated program is named *yylex*. The *yylex* program will recognize expressions in a stream (called *input* in this memo) and perform the specified actions for each expression as it is detected. See Figure 1.



An overview of Lex
Figure 1

For a trivial example, consider a program to delete from the input all blanks or tabs at the ends of lines.

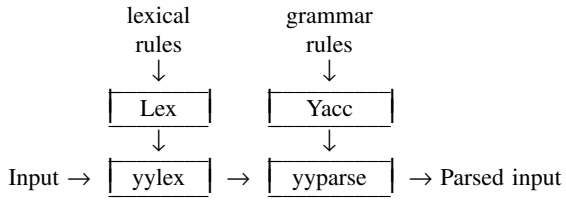
```
%%
[ \t]+$     ;
```

is all that is required. The program contains a %% delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression which matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C language convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates "one or more ..."; and the $ indicates "end of line," as in QED. No action is specified, so the program generated by Lex (yylex) will ignore these characters. Everything else will be copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

```
%%
[ \t]+$     ;
[ \t]+      printf(" ");
```

The finite automaton generated for this source will scan for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a newline character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule all remaining strings of blanks or tabs.

Lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. Lex can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface Lex and Yacc [3]. Lex programs recognize only regular expressions; Yacc writes parsers that accept a large class of context free grammars, but require a lower level analyzer to recognize input tokens. Thus, a combination of Lex and Yacc is often appropriate. When used as a preprocessor for a later parser generator, Lex is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. The flow of control in such a case (which might be the first half of a compiler, for example) is shown in Figure 2. Additional programs, written by other generators or by hand, can be added easily to programs written by Lex.



Lex with Yacc
Figure 2

Yacc users will realize that the name *yylex* is what Yacc expects its lexical analyzer to be named, so that the use of this name by Lex simplifies interfacing.

Lex generates a deterministic finite automaton from the regular expressions in the source [4]. The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time taken by a Lex program to recognize and partition an input stream is proportional to the length of the input. The number of Lex rules or the complexity of the rules is not important in determining speed, unless rules which include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by Lex.

In the program written by Lex, the user's fragments (representing the *actions* to be performed as each regular expression is found) are gathered as

cases of a switch. The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.

Lex is not limited to source which can be interpreted on the basis of one character lookahead. For example, if there are two rules, one looking for *ab* and another for *abcdefg*, and the input stream is *abcdefh*, Lex will recognize *ab* and leave the input pointer just before *cd*. . . Such backup is more costly than the processing of simpler languages.

2. Lex Source.

The general format of Lex source is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where the definitions and the user subroutines are often omitted. The second *%%* is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus

```
%%
```

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

In the outline of Lex programs shown above, the *rules* represent the user's control decisions; they are a table, in which the left column contains *regular expressions* (see section 3) and the right column contains *actions,* program fragments to be executed when the expressions are recognized. Thus an individual rule might appear

```
integer    printf("found keyword INT");
```

to look for the string *integer* in the input stream and print the message "found keyword INT" whenever it appears. In this example the host procedural language is C and the C library function *printf* is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces. As a slightly more useful example, suppose it is desired to change a number of words from British to American spelling. Lex rules such as

```
colour        printf("color");
mechanise     printf("mechanize");
petrol        printf("gas");
```

would be a start. These rules are not quite enough, since the word *petroleum* would become *gaseum*; a way of dealing with this will be described later.

3. Lex Regular Expressions.

The definitions of regular expressions are very similar to those in QED [5]. A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; thus the regular expression

```
integer
```

matches the string *integer* wherever it appears and the expression

```
a57D
```

looks for the string *a57D.*

Operators. The operator characters are

```
" \ [ ] ^ – ? . * + | ( ) $ / { } % < >
```

and if they are to be used as text characters, an escape should be used. The quotation mark operator (") indicates that whatever is contained between a pair of quotes is to be taken as text characters. Thus

```
xyz"++"
```

matches the string *xyz++* when it appears. Note that a part of a string may be quoted. It is harmless but unnecessary to quote an ordinary text character; the expression

```
"xyz++"
```

is the same as the one above. Thus by quoting every non-alphanumeric character being used as a text character, the user can avoid remembering the list above of current operator characters, and is safe should further extensions to Lex lengthen the list.

An operator character may also be turned into a text character by preceding it with \ as in

```
xyz\+\+
```

which is another, less readable, equivalent of the above expressions. Another use of the quoting mechanism is to get a blank into an expression; normally, as explained above, blanks or tabs end a rule. Any blank character not contained within [ ] (see below) must be quoted. Several normal C escapes with \ are recognized: \n is newline, \t is tab, and \b is backspace. To enter \ itself, use \\. Since newline is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.

Character classes. Classes of characters can be specified using the operator pair [ ]. The construction *[abc]* matches a single character, which may be *a*, *b*, or *c*. Within square brackets, most operator meanings are ignored. Only three characters are special: these are \ – and ^. The – charac-

ter indicates ranges. For example,

[a–z0–9<>_]

indicates the character class containing all the lower case letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using – between any pair of characters which are not both upper case letters, both lower case letters, or both digits is implementation dependent and will get a warning message. (E.g., [0–z] in ASCII is many more characters than it is in EBCDIC). If it is desired to include the character – in a character class, it should be first or last; thus

[–+0–9]

matches all the digits and the two signs.

In character classes, the ^ operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the computer character set. Thus

[^abc]

matches all characters except a, b, or c, including all special or control characters; or

[^a–zA–Z]

is any character which is not a letter. The \ character provides the usual escapes within character class brackets.

Arbitrary character. To match almost any character, the operator character

.

is the class of all characters except newline. Escaping into octal is possible although non-portable:

[\40–\176]

matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

Optional expressions. The operator *?* indicates an optional element of an expression. Thus

ab?c

matches either *ac* or *abc*.

Repeated expressions. Repetitions of classes are indicated by the operators * and +.

*a**

is any number of consecutive *a* characters, including zero; while

a+

is one or more instances of *a.* For example,

[a–z]+

is all strings of lower case letters. And

[A–Za–z][A–Za–z0–9]*

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

Alternation and Grouping. The operator | indicates alternation:

(ab|cd)

matches either *ab* or *cd.* Note that parentheses are used for grouping, although they are not necessary on the outside level;

ab|cd

would have sufficed. Parentheses can be used for more complex expressions:

(ab|cd+)?(ef)*

matches such strings as *abefef*, *efefef*, *cdef*, or *cddd*; but not *abc*, *abcd*, or *abcdef*.

Context sensitivity. Lex will recognize a small amount of surrounding context. The two simplest operators for this are ^ and *$*. If the first character of an expression is ^, the expression will only be matched at the beginning of a line (after a newline character, or at the beginning of the input stream). This can never conflict with the other meaning of ^, complementation of character classes, since that only applies within the [ ] operators. If the very last character is *$*, the expression will only be matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the / operator character, which indicates trailing context. The expression

ab/cd

matches the string *ab*, but only if followed by *cd.* Thus

ab$

is the same as

ab/\n

Left context is handled in Lex by *start conditions* as explained in section 10. If a rule is only to be executed when the Lex automaton interpreter is in start condition *x,* the rule should be prefixed by

<x>

using the angle bracket operator characters. If we considered ‘‘being at the beginning of a line’’ to be start condition *ONE*, then the ^ operator would be equivalent to

<ONE>

Start conditions are explained more fully later.

Repetitions and Definitions. The operators {} specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example

{digit}

looks for a predefined string named *digit* and inserts it at that point in the expression. The definitions are given in the first part of the Lex input, before the rules. In contrast,

a{1,5}

looks for 1 to 5 occurrences of *a*.

Finally, initial *%* is special, being the separator for Lex source segments.

4. Lex Actions.

When an expression written as above is matched, Lex executes the corresponding action. This section describes some features of Lex which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus the Lex user who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When Lex is being used with Yacc, this is the normal situation. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted. Also, a character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

One of the simplest things that can be done is to ignore the input. Specifying a C null statement, *;* as an action causes this result. A frequent rule is

```
[ \t\n]     ;
```

which causes the three spacing characters (blank, tab, and newline) to be ignored.

Another easy way to avoid writing actions is the action character |, which indicates that the action for this rule is the action for the next rule. The previous example could also have been written

```
" "
"\t"
"\n"
```

with the same result, although in different style. The quotes around \n and \t are not required.

In more complex actions, the user will often want to know the actual text that matched some expression like *[a–z]+*. Lex leaves this text in an external character array named *yytext.* Thus, to print the name found, a rule like

```
[a–z]+     printf("%s", yytext);
```

will print the string in *yytext.* The C function *printf* accepts a format argument and data to be printed; in this case, the format is ‘‘print string’’ (% indicating data conversion, and *s* indicating string type), and the data are the characters in *yytext.* So this just places the matched string on the output. This action is so common that it may be written as ECHO:

```
[a–z]+     ECHO;
```

is the same as the above. Since the default action is just to print the characters found, one might ask why give a rule, like this one, which merely specifies the default action? Such rules are often required to avoid matching some other rule which is not desired. For example, if there is a rule which matches *read* it will normally match the instances of *read* contained in *bread* or *readjust*; to avoid this, a rule of the form *[a–z]+* is needed. This is explained further below.

Sometimes it is more convenient to know the end of what has been found; hence Lex also provides a count *yyleng* of the number of characters matched. To count both the number of words and the number of characters in words in the input, the user might write

```
[a–zA–Z]+     {words++; chars += yyleng;}
```

which accumulates in *chars* the number of characters in the words recognized. The last character in the string matched can be accessed by

```
yytext[yyleng–1]
```

Occasionally, a Lex action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, *yymore()* can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string would overwrite the current entry in *yytext.* Second, *yyless (n)* may be called to indicate that not all the characters matched by the currently successful expression are wanted right now. The argument *n* indicates the number of characters in *yytext* to be retained. Further characters previously matched are returned to the input. This provides the same sort of lookahead offered by the / operator, but in a different form.

Example: Consider a language which defines a string as a set of characters between quotation (") marks, and provides that to include a " in a string it must be preceded by a \. The regular expression which matches that is somewhat confusing, so that it might be preferable to write

```
\"[^"]*     {
            if (yytext[yyleng–1] == '\\')
                yymore();
            else
                ... normal user processing
            }
```

which will, when faced with a string such as *"abc\"def"* first match the five characters *"abc*; then the call to *yymore()* will cause the next part of the string, *"def*, to be tacked on the end. Note that the final quote terminating the string should be picked up in the code labeled ‘‘normal processing’’.

The function *yyless()* might be used to reprocess text in various circumstances. Consider the C problem of distinguishing the ambiguity of ‘‘=–a’’. Suppose it is desired to treat this as ‘‘=– a’’ but print a message. A rule might be

```
=–[a–zA–Z]    {
              printf("Op (=–) ambiguous\n");
              yyless(yyleng–1);
              ... action for =– ...
              }
```

which prints a message, returns the letter after the operator to the input stream, and treats the operator as ''=–''. Alternatively it might be desired to treat this as ''= –a''. To do this, just return the minus sign as well as the letter to the input:

```
=–[a–zA–Z]    {
              printf("Op (=–) ambiguous\n");
              yyless(yyleng–2);
              ... action for = ...
              }
```

will perform the other interpretation. Note that the expressions for the two cases might more easily be written

=–/[A–Za–z]

in the first case and

=/–[A–Za–z]

in the second; no backup would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of ''=–3'', however, makes

=–/[^ \t\n]

a still better rule.

In addition to these routines, Lex also permits access to the I/O routines it uses. They are:

1) *input()* which returns the next input character;
2) *output(c)* which writes the character *c* on the output; and
3) *unput(c)* pushes the character *c* back onto the input stream to be read later by *input()*.

By default these routines are provided as macro definitions, but the user can override them and supply private versions. These routines define the relationship between external files and internal characters, and must all be retained or modified consistently. They may be redefined, to cause input or output to be transmitted to or from strange places, including other programs or internal memory; but the character set used must be consistent in all routines; a value of zero returned by *input* must mean end of file; and the relationship between *unput* and *input* must be retained or the Lex lookahead will not work. Lex does not look ahead at all if it does not have to, but every rule ending in + * ? or $ or containing / implies lookahead. Lookahead is also necessary to match an expression that is a prefix of another expression. See below for a discussion of the character set used by Lex. The standard Lex library imposes a 100 character limit on backup.

Another Lex library routine that the user will sometimes want to redefine is *yywrap()* which is called whenever Lex reaches an end-of-file. If *yywrap* returns a 1, Lex continues with the normal wrapup on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new source. In this case, the user should provide a *yywrap* which arranges for new input and returns 0. This instructs Lex to continue processing. The default *yywrap* always returns 1.

This routine is also a convenient place to print tables, summaries, etc. at the end of a program. Note that it is not possible to write a normal rule which recognizes end-of-file; the only access to this condition is through *yywrap*. In fact, unless a private version of *input()* is supplied a file containing nulls cannot be handled, since a value of 0 returned by *input* is taken to be end-of-file.

5. Ambiguous Source Rules.

Lex can handle ambiguous specifications. When more than one expression can match the current input, Lex chooses as follows:

1) The longest match is preferred.
2) Among rules which matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules

```
integer    keyword action ...;
[a–z]+     identifier action ...;
```

to be given in that order. If the input is *integers*, it is taken as an identifier, because *[a–z]+* matches 8 characters while *integer* matches only 7. If the input is *integer*, both rules match 7 characters, and the keyword rule is selected because it was given first. Anything shorter (e.g. *int*) will not match the expression *integer* and so the identifier interpretation is used.

The principle of preferring the longest match makes rules containing expressions like .* dangerous. For example,

'.*'

might seem a good way of recognizing a string in single quotes. But it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input

'first' quoted string here, 'second' here

the above expression will match

'first' quoted string here, 'second'

which is probably not what was wanted. A better rule is of the form

'[^'\n]*'

which, on the above input, will stop after *'first'*. The consequences of errors like this are mitigated by

the fact that the . operator will not match newline. Thus expressions like .* stop on the current line. Don't try to defeat this with expressions like *(.|\n)+* or equivalents; the Lex generated program will try to read the entire input file, causing internal buffer overflows.

Note that Lex is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for once and only once. For example, suppose it is desired to count occurrences of both ***she*** and ***he*** in an input text. Some Lex rules to do this might be

```
she     s++;
he      h++;
\n      |
.       ;
```

where the last two rules ignore everything besides ***he*** and ***she***. Remember that . does not include newline. Since ***she*** includes ***he***, Lex will normally ***not*** recognize the instances of ***he*** included in ***she***, since once it has passed a ***she*** those characters are gone.

Sometimes the user would like to override this choice. The action REJECT means "go do the next alternative." It causes whatever rule was second choice after the current rule to be executed. The position of the input pointer is adjusted accordingly. Suppose the user really wants to count the included instances of ***he***:

```
she     {s++; REJECT;}
he      {h++; REJECT;}
\n      |
.       ;
```

these rules are one way of changing the previous example to do just that. After counting each expression, it is rejected; whenever appropriate, the other expression will then be counted. In this example, of course, the user could note that ***she*** includes ***he*** but not vice versa, and omit the REJECT action on ***he***; in other cases, however, it would not be possible a priori to tell which input characters were in both classes.

Consider the two rules

```
a[bc]+      { ... ; REJECT;}
a[cd]+      { ... ; REJECT;}
```

If the input is ***ab***, only the first rule matches, and on ***ad*** only the second matches. The input string ***accb*** matches the first rule for four characters and then the second rule for three characters. In contrast, the input ***accd*** agrees with the second rule for four characters and then the first rule for three.

In general, REJECT is useful whenever the purpose of Lex is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other. Suppose a digram table of the input is desired; normally the digrams overlap, that is the word ***the*** is considered to contain both ***th*** and ***he***. Assuming a two-dimensional array named ***digram*** to be incremented, the appropriate source is

```
%%
[a–z][a–z]      {
                digram[yytext[0]][yytext[1]]++;
                REJECT;
                }
.               ;
\n              ;
```

where the REJECT is necessary to pick up a letter pair beginning at every character, rather than at every other character.

6. Lex Source Definitions.

Remember the format of the Lex source:

```
{definitions}
%%
{rules}
%%
{user routines}
```

So far only the rules have been described. The user needs additional options, though, to define variables for use in his program and for use by Lex. These can go either in the definitions section or in the rules section.

Remember that Lex is turning the rules into a program. Any source not intercepted by Lex is copied into the generated program. There are three classes of such things.

1) Any line which is not part of a Lex rule or action which begins with a blank or tab is copied into the Lex generated program. Such source input prior to the first %% delimiter will be external to any function in the code; if it appears immediately after the first %%, it appears in an appropriate place for declarations in the function written by Lex which contains the actions. This material must look like program fragments, and should precede the first Lex rule.

   As a side effect of the above, lines which begin with a blank or tab, and which contain a comment, are passed through to the generated program. This can be used to include comments in either the Lex source or the generated code. The comments should follow the host language convention.

2) Anything included between lines containing only *%{* and *%}* is copied out as above. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column 1, or copying lines that do not look like programs.

3) Anything after the third %% delimiter, regardless of formats, etc., is copied out after the Lex output.

Definitions intended for Lex are given before the first %% delimiter. Any line in this section not contained between %{ and %}, and begining in column 1, is assumed to define Lex substitution strings. The format of such lines is

name translation

and it causes the string given as a translation to be associated with the name. The name and translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be called out by the {name} syntax in a rule. Using {D} for the digits and {E} for an exponent field, for example, might abbreviate rules to recognize numbers:

```
D                       [0–9]
E                       [DEde][–+]?{D}+
%%
{D}+                    printf("integer");
{D}+"."{D}*({E})?       |
{D}*"."{D}+({E})?       |
{D}+{E}
```

Note the first two rules for real numbers; both require a decimal point and contain an optional exponent field, but the first requires at least one digit before the decimal point and the second requires at least one digit after the decimal point. To correctly handle the problem posed by a Fortran expression such as *35.EQ.I*, which does not contain a real number, a context-sensitive rule such as

```
[0–9]+/"."EQ    printf("integer");
```

could be used in addition to the normal rule for integers.

The definitions section may also contain other commands, including the selection of a host language, a character set table, a list of start conditions, or adjustments to the default size of arrays within Lex itself for larger source programs. These possibilities are discussed below under ‘‘Summary of Source Format,’’ section 12.

7. Usage.

There are two steps in compiling a Lex source program. First, the Lex source must be turned into a generated program in the host general purpose language. Then this program must be compiled and loaded, usually with a library of Lex subroutines. The generated program is on a file named *lex.yy.c*. The I/O library is defined in terms of the C standard library [6].

The C programs generated by Lex are slightly different on OS/370, because the OS compiler is less powerful than the UNIX or GCOS compilers, and does less at compile time. C programs generated on GCOS and UNIX are the same.

UNIX. The library is accessed by the loader flag *–ll*. So an appropriate set of commands is

```
lex source cc lex.yy.c –ll
```

The resulting program is placed on the usual file *a.out* for later execution. To use Lex with Yacc see below. Although the default Lex I/O routines use the C standard library, the Lex automata themselves do not do so; if private versions of *input, output* and *unput* are given, the library can be avoided.

8. Lex and Yacc.

If you want to use Lex with Yacc, note that what Lex writes is a program named *yylex(),* the name required by Yacc for its analyzer. Normally, the default main program on the Lex library calls this routine, but if Yacc is loaded, and its main program is used, Yacc will call *yylex().* In this case each Lex rule should end with

```
return(token);
```

where the appropriate token value is returned. An easy way to get access to Yacc’s names for tokens is to compile the Lex output file as part of the Yacc output file by placing the line

```
# include "lex.yy.c"
```

in the last section of Yacc input. Supposing the grammar to be named ‘‘good’’ and the lexical rules to be named ‘‘better’’ the UNIX command sequence can just be:

```
yacc good
lex better
cc y.tab.c –ly –ll
```

The Yacc library (–ly) should be loaded before the Lex library, to obtain a main program which invokes the Yacc parser. The generations of Lex and Yacc programs can be done in either order.

9. Examples.

As a trivial problem, consider copying an input file while adding 3 to every positive number divisible by 7. Here is a suitable Lex source program

```
%%
                int k;
[0–9]+          {
```

```
        k = atoi(yytext);
        if (k%7 == 0)
            printf("%d", k+3);
        else
            printf("%d",k);
        }
```

to do just that. The rule [0–9]+ recognizes strings of digits; *atoi* converts the digits to binary and stores the result in *k*. The operator % (remainder) is used to check whether *k* is divisible by 7; if it is, it is incremented by 3 as it is written out. It may be objected that this program will alter such input items as *49.63* or *X7*. Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as here:

```
%%
                    int k;
–?[0–9]+            {
                    k = atoi(yytext);
                    printf("%d",
                      k%7 == 0 ? k+3 : k);
                    }
–?[0–9.]+           ECHO;
[A-Za-z][A-Za-z0-9]+ ECHO;
```

Numerical strings containing a "." or preceded by a letter will be picked up by one of the last two rules, and not changed. The *if–else* has been replaced by a C conditional expression to save space; the form *a?b:c* means "if *a* then *b* else *c*".

For an example of statistics gathering, here is a program which histograms the lengths of words, where a word is defined as a string of letters.

```
                    int lengs[100];
%%
[a–z]+              lengs[yyleng]++;
.                   |
\n                  ;
%%
yywrap()
{
int i;
printf("Length  No. words\n");
for(i=0; i<100; i++)
    if (lengs[i] > 0)
        printf("%5d%10d\n",i,lengs[i]);
return(1);
}
```

This program accumulates the histogram, while producing no output. At the end of the input it prints the table. The final statement *return(1);* indicates that Lex is to perform wrapup. If *yywrap* returns zero (false) it implies that further input is available and the program is to continue reading and processing. To provide a *yywrap* that never returns true causes an infinite loop.

As a larger example, here are some parts of a program written by N. L. Schryer to convert double precision Fortran to single precision Fortran. Because Fortran does not distinguish upper and lower case letters, this routine begins by defining a set of classes including both cases of each letter:

```
a       [aA]
b       [bB]
c       [cC]
...
z       [zZ]
```

An additional class recognizes white space:

```
W       [ \t]*
```

The first rule changes "double precision" to "real", or "DOUBLE PRECISION" to "REAL".

```
{d}{o}{u}{b}{l}{e}{W}{p}{r}{e}{c}{i}{s}{i}{o}{n} {
    printf(yytext[0]=='d'? "real" : "REAL");
    }
```

Care is taken throughout this program to preserve the case (upper or lower) of the original program. The conditional operator is used to select the proper form of the keyword. The next rule copies continuation card indications to avoid confusing them with constants:

```
^"      "[^ 0]      ECHO;
```

In the regular expression, the quotes surround the blanks. It is interpreted as "beginning of line, then five blanks, then anything but blank or zero." Note the two different meanings of ^. There follow some rules to change double precision constants to ordinary floating constants.

```
[0–9]+{W}{d}{W}[+–]?{W}[0–9]+     |
[0–9]+{W}"."{W}{d}{W}[+–]?{W}[0–9]+     |
"."{W}[0–9]+{W}{d}{W}[+–]?{W}[0–9]+     {
    /* convert constants */
    for(p=yytext; *p != 0; p++)
        {
        if (*p == 'd' || *p == 'D')
            *p=+ 'e'– 'd';
        ECHO;
        }
```

After the floating point constant is recognized, it is scanned by the *for* loop to find the letter *d* or *D*. The program than adds *'e'–'d'*, which converts it to the next letter of the alphabet. The modified constant, now single-precision, is written out again. There follow a series of names which must be respelled to remove their initial *d*. By using the array *yytext* the same action suffices for all the names (only a sample of a rather long list is given here).

```
{d}{s}{i}{n}            |
```

```
{d}{c}{o}{s}          |
{d}{s}{q}{r}{t}       |
{d}{a}{t}{a}{n}       |
...
{d}{f}{l}{o}{a}{t}    printf("%s",yytext+1);
```

Another list of names must have initial ***d*** changed to initial ***a***:

```
{d}{l}{o}{g}       |
{d}{l}{o}{g}10     |
{d}{m}{i}{n}1      |
{d}{m}{a}{x}1      {
                   yytext[0] =+ 'a' – 'd';
                   ECHO;
                   }
```

And one routine must have initial ***d*** changed to initial ***r***:

```
{d}1{m}{a}{c}{h}    {yytext[0] =+ 'r' – 'd';
```

To avoid such names as ***dsinx*** being detected as instances of ***dsin***, some final rules pick up longer words as identifiers and copy some surviving characters:

```
[A–Za–z][A–Za–z0–9]*    |
[0–9]+                  |
\n                      |
.                       ECHO;
```

Note that this program is not complete; it does not deal with the spacing problems in Fortran or with the use of keywords as identifiers.

10. Left Context Sensitivity.

Sometimes it is desirable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The ^ operator, for example, is a prior context operator, recognizing immediately preceding left context just as ***$*** recognizes immediately following right context. Adjacent left context could be extended, to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often the relevant left context appeared some time earlier, such as at the beginning of a line.

This section describes three means of dealing with different environments: a simple use of flags, when only a few rules change from one environment to another, the use of ***start conditions*** on rules, and the possibility of making multiple lexical analyzers all run together. In each case, there are rules which recognize the need to change the environment in which the following input text is analyzed, and set some parameter to reflect the change. This may be a flag explicitly tested by the user's action code; such a flag is the simplest way of dealing with the problem, since Lex is not involved at all. It may be more convenient, however, to have Lex remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It will only be recognized when Lex is in that start condition. The current start condition may be changed at any time. Finally, if the sets of rules for the different environments are very dissimilar, clarity may be best achieved by writing several distinct lexical analyzers, and switching from one to another as desired.

Consider the following problem: copy the input to the output, changing the word ***magic*** to ***first*** on every line which began with the letter ***a***, changing ***magic*** to ***second*** on every line which began with the letter ***b***, and changing ***magic*** to ***third*** on every line which began with the letter ***c***. All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a flag:

```
        int flag;
%%
^a      {flag = 'a'; ECHO;}
^b      {flag = 'b'; ECHO;}
^c      {flag = 'c'; ECHO;}
\n      {flag =  0 ; ECHO;}
magic   {
        switch (flag)
        {
        case 'a': printf("first"); break;
        case 'b': printf("second"); break;
        case 'c': printf("third"); break;
        default: ECHO; break;
        }
        }
```

should be adequate.

To handle the same problem with start conditions, each start condition must be introduced to Lex in the definitions section with a line reading

```
%Start     name1 name2 ...
```

where the conditions may be named in any order. The word ***Start*** may be abbreviated to ***s*** or ***S***. The conditions may be referenced at the head of a rule with the <> brackets:

```
<name1>expression
```

is a rule which is only recognized when Lex is in the start condition ***name1***. To enter a start condition, execute the action statement

```
BEGIN name1;
```

which changes the start condition to ***name1***. To resume the normal state,

```
BEGIN 0;
```

resets the initial condition of the Lex automaton interpreter. A rule may be active in several start conditions:

```
<name1,name2,name3>
```

is a legal prefix. Any rule not beginning with the <> prefix operator is always active.

The same example as before can be written:

```
%START AA BB CC
%%
^a                  {ECHO; BEGIN AA;}
^b                  {ECHO; BEGIN BB;}
^c                  {ECHO; BEGIN CC;}
\n                  {ECHO; BEGIN 0;}
<AA>magic           printf("first");
<BB>magic           printf("second");
<CC>magic           printf("third");
```

where the logic is exactly the same as in the previous method of handling the problem, but Lex does the work rather than the user's code.

11. Character Set.

The programs generated by Lex handle character I/O only through the routines *input, output,* and *unput.* Thus the character representation provided in these routines is accepted by Lex and employed to return values in *yytext.* For internal use a character is represented as a small integer which, if the standard library is used, has a value equal to the integer value of the bit pattern representing the character on the host computer. Normally, the letter *a* is represented as the same form as the character constant *'a'*. If this interpretation is changed, by providing I/O routines which translate the characters, Lex must be told about it, by giving a translation table. This table must be in the definitions section, and must be bracketed by lines containing only "%T". The table contains lines of the form

```
{integer} {character string}
```

which indicate the value associated with each character. Thus the next example

```
%T
 1      Aa
 2      Bb
...
26      Zz
27      \n
28      +
29      –
30      0
31      1
...
39      9
%T
```

Sample character table.

maps the lower and upper case letters together into the integers 1 through 26, newline into 27, + and – into 28 and 29, and the digits into 30 through 39. Note the escape for newline. If a table is supplied, every character that is to appear either in the rules or in any valid input must be included in the table. No character may be assigned the number 0, and no character may be assigned a bigger number than the size of the hardware character set.

12. Summary of Source Format.

The general form of a Lex source file is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

The definitions section contains a combination of

1) Definitions, in the form "name space translation".
2) Included code, in the form "space code".
3) Included code, in the form
```
%{
code
%}
```
4) Start conditions, given in the form
```
%S name1 name2 ...
```
5) Character set tables, in the form
```
%T
number space character-string
...
%T
```
6) Changes to internal array sizes, in the form
```
%x  nnn
```
where *nnn* is a decimal integer representing an array size and *x* selects the parameter as follows:

Letter	Parameter
p	positions
n	states
e	tree nodes
a	transitions
k	packed character classes
o	output array size

Lines in the rules section have the form "expression action" where the action may be continued on succeeding lines by using braces to delimit it.

Regular expressions in Lex use the following operators:

x	the character "x"
"x"	an "x", even if x is an operator.
\x	an "x", even if x is an operator.

[xy]	the character x or y.
[x–z]	the characters x, y or z.
[^x]	any character but x.
.	any character but newline.
^x	an x at the beginning of a line.
<y>x	an x when Lex is in start condition y.
x$	an x at the end of a line.
x?	an optional x.
x*	0,1,2, ... instances of x.
x+	1,2,3, ... instances of x.
x|y	an x or a y.
(x)	an x.
x/y	an x but only if followed by y.
{xx}	the translation of xx from the definitions section.
x{m,n}	*m* through *n* occurrences of x

13. Caveats and Bugs.

There are pathological expressions which produce exponential growth of the tables when converted to deterministic machines; fortunately, they are rare.

REJECT does not rescan the input; instead it remembers the results of the previous scan. This means that if a rule with trailing context is found, and REJECT executed, the user must not have used *unput* to change the characters forthcoming from the input stream. This is the only restriction on the user's ability to manipulate the not-yet-processed input.

14. Acknowledgments.

As should be obvious from the above, the outside of Lex is patterned on Yacc and the inside on Aho's string matching routines. Therefore, both S. C. Johnson and A. V. Aho are really originators of much of Lex, as well as debuggers of it. Many thanks are due to both.

The code of the current version of Lex was designed, written, and debugged by Eric Schmidt.

15. References.

1. B. W. Kernighan and D. M. Ritchie, ***The C Programming Language,*** Prentice-Hall, N. J. (1978).
2. B. W. Kernighan, ***Ratfor: A Preprocessor for a Rational Fortran,*** Software – Practice and Experience, **5**, pp. 395-496 (1975).
3. S. C. Johnson, ***Yacc: Yet Another Compiler Compiler,*** Computing Science Technical Report No. 32, 1975, Bell Laboratories, Murray Hill, NJ 07974.
4. A. V. Aho and M. J. Corasick, ***Efficient String Matching: An Aid to Bibliographic Search,*** Comm. ACM **18,** 333-340 (1975).
5. B. W. Kernighan, D. M. Ritchie and K. L. Thompson, ***QED Text Editor,*** Computing Science Technical Report No. 5, 1972, Bell Laboratories, Murray Hill, NJ 07974.
6. D. M. Ritchie, private communication. See also M. E. Lesk, ***The Portable C Library,*** Computing Science Technical Report No. 31, Bell Laboratories, Murray Hill, NJ 07974.

The M4 Macro Processor

Brian W. Kernighan

Dennis M. Ritchie

ABSTRACT

M4 is a macro processor available on UNIX† and GCOS. Its primary use has been as a front end for Ratfor for those cases where parameterless macros are not adequately powerful. It has also been used for languages as disparate as C and Cobol. M4 is particularly suited for functional languages like Fortran, PL/I and C since macros are specified in a functional notation.

M4 provides features seldom found even in much larger macro processors, including

- arguments
- condition testing
- arithmetic capabilities
- string and substring functions
- file manipulation

This paper is a user's manual for M4.

Introduction

A macro processor is a useful way to enhance a programming language, to make it more palatable or more readable, or to tailor it to a particular application. The **#define** statement in C and the analogous **define** in Ratfor are examples of the basic facility provided by any macro processor — replacement of text by other text.

The M4 macro processor is an extension of a macro processor called M3 which was written by D. M. Ritchie for the AP-3 minicomputer; M3 was in turn based on a macro processor implemented for [1]. Readers unfamiliar with the basic ideas of macro processing may wish to read some of the discussion there.

M4 is a suitable front end for Ratfor and C, and has also been used successfully with Cobol. Besides the straightforward replacement of one string of text by another, it provides macros with arguments, conditional macro expansion, arithmetic, file manipulation, and some specialized string processing functions.

The basic operation of M4 is to copy its input to its output. As the input is read, however, each alphanumeric "token" (that is, string of letters and digits) is checked. If it is the name of a macro, then the name of the macro is replaced by its defining text, and the resulting string is pushed back onto the input to be rescanned. Macros may be called with arguments, in which case the arguments are collected and substituted into the right places in the defining text before it is rescanned.

M4 provides a collection of about twenty built-in macros which perform various useful operations; in addition, the user can define new macros. Built-ins and user-defined macros work exactly the same way, except that some of the built-in macros have side effects on the state of

† UNIX is a registered trademark of The Open Group in the U.S. and other countries.

the process.

Usage

On UNIX, use

m4 [files]

Each argument file is processed in order; if there are no arguments, or if an argument is `–', the standard input is read at that point. The processed text is written on the standard output, which may be captured for subsequent processing with

m4 [files] >outputfile

On GCOS, usage is identical, but the program is called **./m4**.

Defining Macros

The primary built-in function of M4 is **define**, which is used to define new macros. The input

define(name, stuff)

causes the string **name** to be defined as **stuff**. All subsequent occurrences of **name** will be replaced by **stuff**. **name** must be alphanumeric and must begin with a letter (the underscore _ counts as a letter). **stuff** is any text that contains balanced parentheses; it may stretch over multiple lines.

Thus, as a typical example,

```
define(N, 100)
 ...
if (i > N)
```

defines **N** to be 100, and uses this ``symbolic constant'' in a later **if** statement.

The left parenthesis must immediately follow the word **define**, to signal that **define** has arguments. If a macro or built-in name is not followed immediately by `(', it is assumed to have no arguments. This is the situation for **N** above; it is actually a macro with no arguments, and thus when it is used there need be no (...) following it.

You should also notice that a macro name is only recognized as such if it appears surrounded by non-alphanumerics. For example, in

```
define(N, 100)
 ...
if (NNN > 100)
```

the variable **NNN** is absolutely unrelated to the defined macro **N**, even though it contains a lot of **N**'s.

Things may be defined in terms of other things. For example,

```
define(N, 100)
define(M, N)
```

defines both M and N to be 100.

What happens if **N** is redefined? Or, to say it another way, is **M** defined as **N** or as 100? In M4, the latter is true — **M** is 100, so even if **N** subsequently changes, **M** does not.

This behavior arises because M4 expands macro names into their defining text as soon as it possibly can. Here, that means that when the string **N** is seen as the arguments of **define** are being collected, it is immediately replaced by 100; it's just as if you had said

define(M, 100)

in the first place.

If this isn't what you really want, there are two ways out of it. The first, which is specific to this situation, is to interchange the order of the definitions:

```
define(M, N)
define(N, 100)
```

Now **M** is defined to be the string **N**, so when you ask for **M** later, you'll always get the value of **N** at that time (because the **M** will be replaced by **N** which will be replaced by 100).

Quoting

The more general solution is to delay the expansion of the arguments of **define** by *quoting* them. Any text surrounded by the single quotes ` and ´ is not expanded immediately, but has the quotes stripped off. If you say

```
define(N, 100)
define(M, `N´)
```

the quotes around the **N** are stripped off as the argument is being collected, but they have served their purpose, and **M** is defined as the string **N**, not 100. The general rule is that M4 always strips off one level of single quotes whenever it evaluates something. This is true even outside of macros. If you want the word **define** to appear in the output, you have to quote it in the input, as in

`define´ = 1;

As another instance of the same thing, which is a bit more surprising, consider redefining **N**:

```
define(N, 100)
 ...
define(N, 200)
```

Perhaps regrettably, the **N** in the second definition is evaluated as soon as it's seen; that is, it is replaced by 100, so it's as if you had written

```
define(100, 200)
```

This statement is ignored by M4, since you can only define things that look like names, but it obviously doesn't have the effect you wanted. To really redefine **N**, you must delay the evaluation by quoting:

```
define(N, 100)
 ...
define(`N´, 200)
```

In M4, it is often wise to quote the first argument of a macro.

If ` and ´ are not convenient for some reason, the quote characters can be changed with the built-in **changequote**:

```
changequote([, ])
```

makes the new quote characters the left and right brackets. You can restore the original characters with just

```
changequote
```

There are two additional built-ins related to **define**. **undefine** removes the definition of some macro or built-in:

```
undefine(`N´)
```

removes the definition of **N**. (Why are the quotes absolutely necessary?) Built-ins can be removed with **undefine**, as in

```
undefine(`define´)
```

but once you remove one, you can never get it back.

The built-in **ifdef** provides a way to determine if a macro is currently defined. In particular, M4 has pre-defined the names **unix** and **gcos** on the corresponding systems, so you can tell which one you're using:

```
ifdef(`unix´, `define(wordsize,16)´ )
ifdef(`gcos´, `define(wordsize,36)´ )
```

makes a definition appropriate for the particular machine. Don't forget the quotes!

ifdef actually permits three arguments; if the name is undefined, the value of **ifdef** is then the third argument, as in

```
ifdef(`unix´, on UNIX, not on UNIX)
```

Arguments

So far we have discussed the simplest form of macro processing — replacing one string by another (fixed) string. User-defined macros may also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its **define**) any occurrence of **$n** will be replaced by the **n**th argument when the macro is actually used. Thus, the macro **bump**, defined as

```
define(bump, $1 = $1 + 1)
```

generates code to increment its argument by 1:

```
bump(x)
```

is

```
x = x + 1
```

A macro can have as many arguments as you want, but only the first nine are accessible, through **$1** to **$9**. (The macro name itself is **$0**, although that is less commonly used.) Arguments that are not supplied are replaced by null strings, so we can define a macro **cat** which simply concatenates its arguments, like this:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

Thus

```
cat(x, y, z)
```

is equivalent to

```
xyz
```

$4 through **$9** are null, since no corresponding arguments were provided.

Leading unquoted blanks, tabs, or newlines that occur during argument collection are discarded. All other white space is retained. Thus

```
define(a,    b    c)
```

defines **a** to be **b   c**.

Arguments are separated by commas, but parentheses are counted properly, so a comma ``protected'' by parentheses does not terminate

an argument. That is, in

define(a, (b,c))

there are only two arguments; the second is literally **(b,c)**. And of course a bare comma or parenthesis can be inserted by quoting it.

Arithmetic Built-ins

M4 provides two built-in functions for doing arithmetic on integers (only). The simplest is **incr**, which increments its numeric argument by 1. Thus to handle the common programming situation where you want a variable to be defined as ``one more than N'', write

```
define(N, 100)
define(N1, `incr(N)´)
```

Then **N1** is defined as one more than the current value of **N**.

The more general mechanism for arithmetic is a built-in called **eval**, which is capable of arbitrary arithmetic on integers. It provides the operators (in decreasing order of precedence)

```
unary + and −
** or ^       (exponentiation)
*  /  %  (modulus)
+  −
==  !=  <  <=  >  >=
!             (not)
& or &&       (logical and)
| or ||       (logical or)
```

Parentheses may be used to group operations where needed. All the operands of an expression given to **eval** must ultimately be numeric. The numeric value of a true relation (like 1>0) is 1, and false is 0. The precision in **eval** is 32 bits on UNIX and 36 bits on GCOS.

As a simple example, suppose we want **M** to be **2******N+1**. Then

```
define(N, 3)
define(M, `eval(2**N+1)´)
```

As a matter of principle, it is advisable to quote the defining text for a macro unless it is very simple indeed (say just a number); it usually gives the result you want, and is a good habit to get into.

File Manipulation

You can include a new file in the input at any time by the built-in function **include**:

include(filename)

inserts the contents of **filename** in place of the **include** command. The contents of the file is often a set of definitions. The value of **include** (that is, its replacement text) is the contents of the file; this can be captured in definitions, etc.

It is a fatal error if the file named in **include** cannot be accessed. To get some control over this situation, the alternate form **sinclude** can be used; **sinclude** (``silent include'') says nothing and continues if it can't access the file.

It is also possible to divert the output of M4 to temporary files during processing, and output the collected material upon command. M4 maintains nine of these diversions, numbered 1 through 9. If you say

divert(n)

all subsequent output is put onto the end of a temporary file referred to as **n**. Diverting to this file is stopped by another **divert** command; in particular, **divert** or **divert(0)** resumes the normal output process.

Diverted text is normally output all at once at the end of processing, with the diversions output in numeric order. It is possible, however, to bring back diversions at any time, that is, to append them to the current diversion.

undivert

brings back all diversions in numeric order, and **undivert** with arguments brings back the selected diversions in the order given. The act of undiverting discards the diverted stuff, as does diverting into a diversion whose number is not between 0 and 9 inclusive.

The value of **undivert** is *not* the diverted stuff. Furthermore, the diverted material is *not* rescanned for macros.

The built-in **divnum** returns the number of the currently active diversion. This is zero during normal processing.

System Command

You can run any program in the local operating system with the **syscmd** built-in. For example,

syscmd(date)

on UNIX runs the **date** command. Normally **syscmd** would be used to create a file for a sub-

sequent **include**.

To facilitate making unique file names, the built-in **maketemp** is provided, with specifications identical to the system function *mktemp:* a string of XXXXX in the argument is replaced by the process id of the current process.

Conditionals

There is a built-in called **ifelse** which enables you to perform arbitrary conditional testing. In the simplest form,

```
ifelse(a, b, c, d)
```

compares the two strings **a** and **b**. If these are identical, **ifelse** returns the string **c**; otherwise it returns **d**. Thus we might define a macro called **compare** which compares two strings and returns ``yes'' or ``no'' if they are the same or different.

```
define(compare, `ifelse($1, $2, yes, no)´)
```

Note the quotes, which prevent too-early evaluation of **ifelse**.

If the fourth argument is missing, it is treated as empty.

ifelse can actually have any number of arguments, and thus provides a limited form of multi-way decision capability. In the input

```
ifelse(a, b, c, d, e, f, g)
```

if the string **a** matches the string **b**, the result is **c**. Otherwise, if **d** is the same as **e**, the result is **f**. Otherwise the result is **g**. If the final argument is omitted, the result is null, so

```
ifelse(a, b, c)
```

is **c** if **a** matches **b**, and null otherwise.

String Manipulation

The built-in **len** returns the length of the string that makes up its argument. Thus

```
len(abcdef)
```

is 6, and **len((a,b))** is 5.

The built-in **substr** can be used to produce substrings of strings. **substr(s, i, n)** returns the substring of **s** that starts at the **i**th position (origin zero), and is **n** characters long. If **n** is omitted, the rest of the string is returned, so

```
substr(`now is the time´, 1)
```

is

```
ow is the time
```

If **i** or **n** are out of range, various sensible things happen.

index(s1, s2) returns the index (position) in **s1** where the string **s2** occurs, or –1 if it doesn't occur. As with **substr**, the origin for strings is 0.

The built-in **translit** performs character transliteration.

```
translit(s, f, t)
```

modifies **s** by replacing any character found in **f** by the corresponding character of **t**. That is,

```
translit(s, aeiou, 12345)
```

replaces the vowels by the corresponding digits. If **t** is shorter than **f**, characters which don't have an entry in **t** are deleted; as a limiting case, if **t** is not present at all, characters from **f** are deleted from **s**. So

```
translit(s, aeiou)
```

deletes vowels from **s**.

There is also a built-in called **dnl** which deletes all characters that follow it up to and including the next newline; it is useful mainly for throwing away empty lines that otherwise tend to clutter up M4 output. For example, if you say

```
define(N, 100)
define(M, 200)
define(L, 300)
```

the newline at the end of each line is not part of the definition, so it is copied into the output, where it may not be wanted. If you add **dnl** to each of these lines, the newlines will disappear.

Another way to achieve this, due to J. E. Weythman, is

```
divert(–1)
        define(...)
        ...
divert
```

Printing

The built-in **errprint** writes its arguments out on the standard error file. Thus you can say

```
errprint(`fatal error´)
```

dumpdef is a debugging aid which dumps the current definitions of defined terms. If there

are no arguments, you get everything; otherwise you get the ones you name as arguments. Don't forget to quote the names!

Summary of Built-ins

Each entry is preceded by the page number where it is described.

3 changequote(L, R)
1 define(name, replacement)
4 divert(number)
4 divnum
5 dnl
5 dumpdef(`name´, `name´, ...)
5 errprint(s, s, ...)
4 eval(numeric expression)
3 ifdef(`name´, this if true, this if false)
5 ifelse(a, b, c, d)
4 include(file)
3 incr(number)
5 index(s1, s2)
5 len(string)
4 maketemp(...XXXXX...)
4 sinclude(file)
5 substr(string, position, number)
4 syscmd(s)
5 translit(str, from, to)
3 undefine(`name´)
4 undivert(number,number,...)

Acknowledgements

We are indebted to Rick Becker, John Chambers, Doug McIlroy, and especially Jim Weythman, whose pioneering use of M4 has led to several valuable improvements. We are also deeply grateful to Weythman for several substantial contributions to the code.

References

[1] B. W. Kernighan and P. J. Plauger, *Software Tools,* Addison-Wesley, Inc., 1976.

gprof: a Call Graph Execution Profiler[1]

by *Susan L. Graham Peter B. Kessler Marshall K. McKusick*

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

Abstract

Large complex programs are composed of many small routines that implement abstractions for the routines that call them. To be useful, an execution profiler must attribute execution time in a way that is significant for the logical structure of a program as well as for its textual decomposition. This data must then be displayed to the user in a convenient and informative way. The **gprof** profiler accounts for the running time of called routines in the running time of the routines that call them. The design and use of this profiler is described.

1. Programs to be Profiled

Software research environments normally include many large programs both for production use and for experimental investigation. These programs are typically modular, in accordance with generally accepted principles of good program design. Often they consist of numerous small routines that implement various abstractions. Sometimes such large programs are written by one programmer who has understood the requirements for these abstractions, and has programmed them appropriately. More frequently the program has had multiple authors and has evolved over time, changing the demands placed on the implementation of the abstractions without changing the implementation itself. Finally, the program may be assembled from a library of abstraction implementations unexamined by the programmer.

Once a large program is executable, it is often desirable to increase its speed, especially if small portions of the program are found to dominate its execution time. The purpose of the **gprof** profiling tool is to help the user evaluate alternative implementations of abstractions. We developed this tool in response to our efforts to improve a code generator we were writing [Graham82].

The **gprof** design takes advantage of the fact that the programs to be measured are large, structured and hierarchical. We provide a profile in which the execution time for a set of routines that implement an abstraction is collected and charged to that abstraction. The profile can be used to compare and assess the costs of various implementations.

The profiler can be linked into a program without special planning by the programmer. The overhead for using **gprof** is low; both in terms of added execution time and in the volume of profiling information recorded.

2. Types of Profiling

There are several different uses for program profiles, and each may require different information from the profiles, or different presentation of the information. We distinguish two broad categories of profiles: those that present counts of statement or routine invocations, and those that display timing information about statements or routines. Counts are typically presented in tabular form, often in parallel with a listing of the source code. Timing information could be similarly presented; but more than one measure of time might be associated with each statement or routine. For example, in the framework used by **gprof** each profiled segment would display two times: one for the time used by the segment

[1]This work was supported by grant MCS80-05144 from the National Science Foundation.

itself, and another for the time inherited from code segments it invokes.

Execution counts are used in many different contexts. The exact number of times a routine or statement is activated can be used to determine if an algorithm is performing as expected. Cursory inspection of such counters may show algorithms whose complexity is unsuited to the task at hand. Careful interpretation of counters can often suggest improvements to acceptable algorithms. Precise examination can uncover subtle errors in an algorithm. At this level, profiling counters are similar to debugging statements whose purpose is to show the number of times a piece of code is executed. Another view of such counters is as boolean values. One may be interested that a portion of code has executed at all, for exhaustive testing, or to check that one implementation of an abstraction completely replaces a previous one.

Execution counts are not necessarily proportional to the amount of time required to execute the routine or statement. Further, the execution time of a routine will not be the same for all calls on the routine. The criteria for establishing execution time must be decided. If a routine implements an abstraction by invoking other abstractions, the time spent in the routine will not accurately reflect the time required by the abstraction it implements. Similarly, if an abstraction is implemented by several routines the time required by the abstraction will be distributed across those routines.

Given the execution time of individual routines, **gprof** accounts to each routine the time spent for it by the routines it invokes. This accounting is done by assembling a *call graph* with nodes that are the routines of the program and directed arcs that represent calls from call sites to routines. We distinguish among three different call graphs for a program. The *complete call graph* incorporates all routines and all potential arcs, including arcs that represent calls to functional parameters or functional variables. This graph contains the other two graphs as subgraphs. The *static call graph* includes all routines and all possible arcs that are not calls to functional parameters or variables. The *dynamic call graph* includes only those routines and arcs traversed by the profiled execution of the program. This graph need not include all routines, nor need it include all potential arcs between the routines it covers. It may, however, include arcs to functional parameters or variables that the static call graph may omit. The static call graph can be determined from the (static) program text. The dynamic call graph is determined only by profiling an execution of the program. The complete call graph for a monolithic program could be determined by data flow analysis techniques. The complete call graph for programs that change during execution, by modifying themselves or dynamically loading or overlaying code, may never be determinable. Both the static call graph and the dynamic call graph are used by **gprof**, but it does not search for the complete call graph.

3. Gathering Profile Data

Routine calls or statement executions can be measured by having a compiler augment the code at strategic points. The additions can be inline increments to counters [Knuth71] [Satterthwaite72] [Joy79] or calls to monitoring routines [Unix]. The counter increment overhead is low, and is suitable for profiling statements. A call of the monitoring routine has an overhead comparable with a call of a regular routine, and is therefore only suited to profiling on a routine by routine basis. However, the monitoring routine solution has certain advantages. Whatever counters are needed by the monitoring routine can be managed by the monitoring routine itself, rather than being distributed around the code. In particular, a monitoring routine can easily be called from separately compiled programs. In addition, different monitoring routines can be linked into the program being measured to assemble different profiling data without having to change the compiler or recompile the program. We have exploited this approach; our compilers for C, Fortran77, and Pascal can insert calls to a monitoring routine in the prologue for each routine. Use of the monitoring routine requires no planning on part of a programmer other than to request that augmented routine prologues be produced during compilation.

We are interested in gathering three pieces of information during program execution: call counts and execution times for each profiled routine, and the arcs of the dynamic call graph traversed by this execution of the program. By post-processing of this data we can build the dynamic call graph for this execution of the program and propagate times along the edges of this graph to attribute times for

routines to the routines that invoke them.

Gathering of the profiling information should not greatly interfere with the running of the program. Thus, the monitoring routine must not produce trace output each time it is invoked. The volume of data thus produced would be unmanageably large, and the time required to record it would overwhelm the running time of most programs. Similarly, the monitoring routine can not do the analysis of the profiling data (e.g. assembling the call graph, propagating times around it, discovering cycles, etc.) during program execution. Our solution is to gather profiling data in memory during program execution and to condense it to a file as the profiled program exits. This file is then processed by a separate program to produce the listing of the profile data. An advantage of this approach is that the profile data for several executions of a program can be combined by the post-processing to provide a profile of many executions.

The execution time monitoring consists of three parts. The first part allocates and initializes the runtime monitoring data structures before the program begins execution. The second part is the monitoring routine invoked from the prologue of each profiled routine. The third part condenses the data structures and writes them to a file as the program terminates. The monitoring routine is discussed in detail in the following sections.

3.1. Execution Counts

The **gprof** monitoring routine counts the number of times each profiled routine is called. The monitoring routine also records the arc in the call graph that activated the profiled routine. The count is associated with the arc in the call graph rather than with the routine. Call counts for routines can then be determined by summing the counts on arcs directed into that routine. In a machine-dependent fashion, the monitoring routine notes its own return address. This address is in the prologue of some profiled routine that is the destination of an arc in the dynamic call graph. The monitoring routine also discovers the return address for that routine, thus identifying the call site, or source of the arc. The source of the arc is in the *caller*, and the destination is in the *callee*. For example, if a routine A calls a routine B, A is the caller, and B is the callee. The prologue of B will include a call to the monitoring routine that will note the arc from A to B and either initialize or increment a counter for that arc.

One can not afford to have the monitoring routine output tracing information as each arc is identified. Therefore, the monitoring routine maintains a table of all the arcs discovered, with counts of the numbers of times each is traversed during execution. This table is accessed once per routine call. Access to it must be as fast as possible so as not to overwhelm the time required to execute the program.

Our solution is to access the table through a hash table. We use the call site as the primary key with the callee address being the secondary key. Since each call site typically calls only one callee, we can reduce (usually to one) the number of minor lookups based on the callee. Another alternative would use the callee as the primary key and the call site as the secondary key. Such an organization has the advantage of associating callers with callees, at the expense of longer lookups in the monitoring routine. We are fortunate to be running in a virtual memory environment, and (for the sake of speed) were able to allocate enough space for the primary hash table to allow a one-to-one mapping from call site addresses to the primary hash table. Thus our hash function is trivial to calculate and collisions occur only for call sites that call multiple destinations (e.g. functional parameters and functional variables). A one level hash function using both call site and callee would result in an unreasonably large hash table. Further, the number of dynamic call sites and callees is not known during execution of the profiled program.

Not all callers and callees can be identified by the monitoring routine. Routines that were compiled without the profiling augmentations will not call the monitoring routine as part of their prologue, and thus no arcs will be recorded whose destinations are in these routines. One need not profile all the routines in a program. Routines that are not profiled run at full speed. Certain routines, notably exception handlers, are invoked by non-standard calling sequences. Thus the monitoring routine may know the destination of an arc (the callee), but find it difficult or impossible to determine the source of the arc

(the caller). Often in these cases the apparent source of the arc is not a call site at all. Such anomalous invocations are declared ‘‘spontaneous’’.

3.2. Execution Times

The execution times for routines can be gathered in at least two ways. One method measures the execution time of a routine by measuring the elapsed time from routine entry to routine exit. Unfortunately, time measurement is complicated on time-sharing systems by the time-slicing of the program. A second method samples the value of the program counter at some interval, and infers execution time from the distribution of the samples within the program. This technique is particularly suited to time-sharing systems, where the time-slicing can serve as the basis for sampling the program counter. Notice that, whereas the first method could provide exact timings, the second is inherently a statistical approximation.

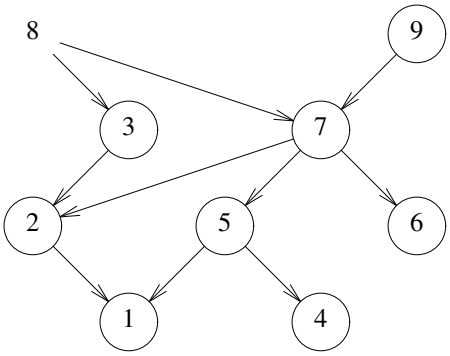
The sampling method need not require support from the operating system: all that is needed is the ability to set and respond to ‘‘alarm clock’’ interrupts that run relative to program time. It is imperative that the intervals be uniform since the sampling of the program counter rather than the duration of the interval is the basis of the distribution. If sampling is done too often, the interruptions to sample the program counter will overwhelm the running of the profiled program. On the other hand, the program must run for enough sampled intervals that the distribution of the samples accurately represents the distribution of time for the execution of the program. As with routine call tracing, the monitoring routine can not afford to output information for each program counter sample. In our computing environment, the operating system can provide a histogram of the location of the program counter at the end of each clock tick (1/60th of a second) in which a program runs. The histogram is assembled in memory as the program runs. This facility is enabled by our monitoring routine. We have adjusted the granularity of the histogram so that program counter values map one-to-one onto the histogram. We make the simplifying assumption that all calls to a specific routine require the same amount of time to execute. This assumption may disguise that some calls (or worse, some call sites) always invoke a routine such that its execution is faster (or slower) than the average time for that routine.

When the profiled program terminates, the arc table and the histogram of program counter samples is written to a file. The arc table is condensed to consist of the source and destination addresses of the arc and the count of the number of times the arc was traversed by this execution of the program. The recorded histogram consists of counters of the number of times the program counter was found to be in each of the ranges covered by the histogram. The ranges themselves are summarized as a lower and upper bound and a step size.

4. Post Processing

Having gathered the arcs of the call graph and timing information for an execution of the program, we are interested in attributing the time for each routine to the routines that call it. We build a dynamic call graph with arcs from caller to callee, and propagate time from descendants to ancestors by topologically sorting the call graph. Time propagation is performed from the leaves of the call graph toward the roots, according to the order assigned by a topological numbering algorithm. The topological numbering ensures that all edges in the graph go from higher numbered nodes to lower numbered nodes. An example is given in Figure 1. If we propagate time from nodes in the order assigned by the algorithm, execution time can be propagated from descendants to ancestors after a single traversal of each arc in the call graph. Each parent receives some fraction of a child’s time. Thus time is charged to the caller in addition to being charged to the callee.

Let C_e be the number of calls to some routine, e, and C_e^r be the number of calls from a caller r to a callee e. Since we are assuming each call to a routine takes the average amount of time for all calls to that routine, the caller is accountable for C_e^r/C_e of the time spent by the callee. Let the S_e be the $selftime$ of a routine, e. The selftime of a routine can be determined from the timing information gathered during profiled program execution. The total time, T_r, we wish to account to a routine r, is then given by the recurrence equation:



Topological ordering
Figure 1.

$$T_r = S_r + \sum_{r \text{ CALLS } e} T_e \times \frac{C_e^r}{C_e}$$

where r CALLS e is a relation showing all routines e called by a routine r. This relation is easily available from the call graph.

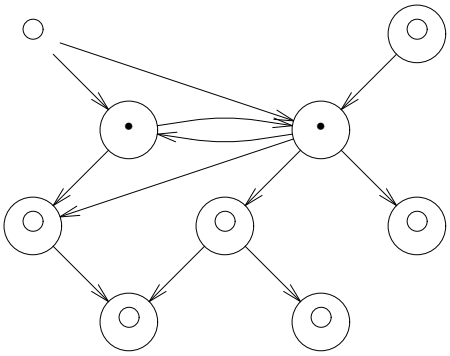
However, if the execution contains recursive calls, the call graph has cycles that cannot be topologically sorted. In these cases, we discover strongly-connected components in the call graph, treat each such component as a single node, and then sort the resulting graph. We use a variation of Tarjan's strongly-connected components algorithm that discovers strongly-connected components as it is assigning topological order numbers [Tarjan72].

Time propagation within strongly connected components is a problem. For example, a self-recursive routine (a trivial cycle in the call graph) is accountable for all the time it uses in all its recursive instantiations. In our scheme, this time should be shared among its call graph parents. The arcs from a routine to itself are of interest, but do not participate in time propagation. Thus the simple equation for time propagation does not work within strongly connected components. Time is not propagated from one member of a cycle to another, since, by definition, this involves propagating time from a routine to itself. In addition, children of one member of a cycle must be considered children of all members of the cycle. Similarly, parents of one member of the cycle must inherit all members of the cycle as descendants. It is for these reasons that we collapse connected components. Our solution collects all members of a cycle together, summing the time and call counts for all members. All calls into the cycle are made to share the total time of the cycle, and all descendants of the cycle propagate time into the cycle as a whole. Calls among the members of the cycle do not propagate any time, though they are listed in the call graph profile.

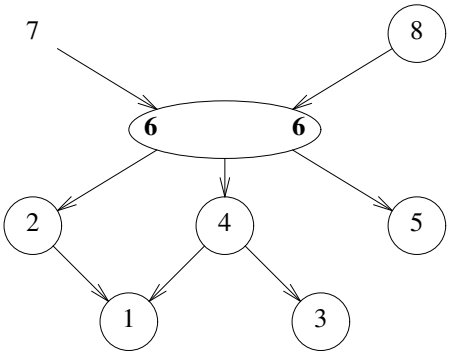
Figure 2 shows a modified version of the call graph of Figure 1, in which the nodes labelled 3 and 7 in Figure 1 are mutually recursive. The topologically sorted graph after the cycle is collapsed is given in Figure 3.

Since the technique described above only collects the dynamic call graph, and the program typically does not call every routine on each execution, different executions can introduce different cycles in the dynamic call graph. Since cycles often have a significant effect on time propagation, it is desirable to incorporate the static call graph so that cycles will have the same members regardless of how the program runs.

The static call graph can be constructed from the source text of the program. However, discovering the static call graph from the source text would require two moderately difficult steps: finding the source text for the program (which may not be available), and scanning and parsing that text, which may be in any one of several languages.



Cycle to be collapsed.
Figure 2.



Topological numbering after cycle collapsing.
Figure 3.

In our programming system, the static calling information is also contained in the executable version of the program, which we already have available, and which is in language-independent form. One can examine the instructions in the object program, looking for calls to routines, and note which routines can be called. This technique allows us to add arcs to those already in the dynamic call graph. If a statically discovered arc already exists in the dynamic call graph, no action is required. Statically discovered arcs that do not exist in the dynamic call graph are added to the graph with a traversal count of zero. Thus they are never responsible for any time propagation. However, they may affect the structure of the graph. Since they may complete strongly connected components, the static call graph construction is done before topological ordering.

5. Data Presentation

The data is presented to the user in two different formats. The first presentation simply lists the routines without regard to the amount of time their descendants use. The second presentation incorporates the call graph of the program.

5.1. The Flat Profile

The flat profile consists of a list of all the routines that are called during execution of the program, with the count of the number of times they are called and the number of seconds of execution time for which they are themselves accountable. The routines are listed in decreasing order of execution time. A list of the routines that are never called during execution of the program is also available to verify that nothing important is omitted by this execution. The flat profile gives a quick overview of the routines that are used, and shows the routines that are themselves responsible for large fractions of the execution time. In practice, this profile usually shows that no single function is overwhelmingly responsible for the total time of the program. Notice that for this profile, the individual times sum to the total execution

time.

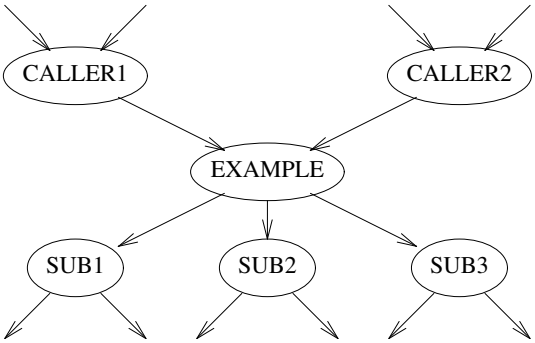
5.2. The Call Graph Profile

Ideally, we would like to print the call graph of the program, but we are limited by the two-dimensional nature of our output devices. We cannot assume that a call graph is planar, and even if it is, that we can print a planar version of it. Instead, we choose to list each routine, together with information about the routines that are its direct parents and children. This listing presents a window into the call graph. Based on our experience, both parent information and child information is important, and should be available without searching through the output.

The major entries of the call graph profile are the entries from the flat profile, augmented by the time propagated to each routine from its descendants. This profile is sorted by the sum of the time for the routine itself plus the time inherited from its descendants. The profile shows which of the higher level routines spend large portions of the total execution time in the routines that they call. For each routine, we show the amount of time passed by each child to the routine, which includes time for the child itself and for the descendants of the child (and thus the descendants of the routine). We also show the percentage these times represent of the total time accounted to the child. Similarly, the parents of each routine are listed, along with time, and percentage of total routine time, propagated to each one.

Cycles are handled as single entities. The cycle as a whole is shown as though it were a single routine, except that members of the cycle are listed in place of the children. Although the number of calls of each member from within the cycle are shown, they do not affect time propagation. When a child is a member of a cycle, the time shown is the appropriate fraction of the time for the whole cycle. Self-recursive routines have their calls broken down into calls from the outside and self-recursive calls. Only the outside calls affect the propagation of time.

The following example is a typical fragment of a call graph.



The entry in the call graph profile listing for this example is shown in Figure 4.

The entry is for routine EXAMPLE, which has the Caller routines as its parents, and the Sub routines as its children. The reader should keep in mind that all information is given *with respect to EXAMPLE*. The index in the first column shows that EXAMPLE is the second entry in the profile listing. The EXAMPLE routine is called ten times, four times by CALLER1, and six times by CALLER2. Consequently 40% of EXAMPLE's time is propagated to CALLER1, and 60% of EXAMPLE's time is propagated to CALLER2. The self and descendant fields of the parents show the amount of self and descendant time EXAMPLE propagates to them (but not the time used by the parents directly). Note that EXAMPLE calls itself recursively four times. The routine EXAMPLE calls routine SUB1 twenty times, SUB2 once, and never calls SUB3. Since SUB2 is called a total of five times, 20% of its self and descendant time is propagated to EXAMPLE's descendant time field. Because SUB1 is a member of *cycle 1*, the self and descendant times and call count fraction are those for the cycle as a whole. Since cycle 1 is called a total of forty times (not counting calls among members of the cycle), it propagates 50% of the cycle's self and descendant time to EXAMPLE's descendant time field. Finally each name is followed by an index that shows where on the listing to find the entry for that routine.

6. Using the Profiles

The profiler is a useful tool for improving a set of routines that implement an abstraction. It can be helpful in identifying poorly coded routines, and in evaluating the new algorithms and code that replace them. Taking full advantage of the profiler requires a careful examination of the call graph profile, and a thorough knowledge of the abstractions underlying the program.

The easiest optimization that can be performed is a small change to a control construct or data structure that improves the running time of the program. An obvious starting point is a routine that is called many times. For example, suppose an output routine is the only parent of a routine that formats the data. If this format routine is expanded inline in the output routine, the overhead of a function call and return can be saved for each datum that needs to be formatted.

The drawback to inline expansion is that the data abstractions in the program may become less parameterized, hence less clearly defined. The profiling will also become less useful since the loss of routines will make its output more granular. For example, if the symbol table functions ‘‘lookup’’, ‘‘insert’’, and ‘‘delete’’ are all merged into a single parameterized routine, it will be impossible to determine the costs of any one of these individual functions from the profile.

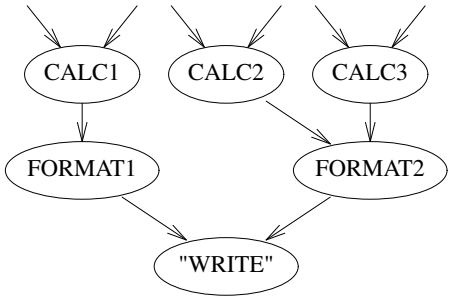
Further potential for optimization lies in routines that implement data abstractions whose total execution time is long. For example, a lookup routine might be called only a few times, but use an inefficient linear search algorithm, that might be replaced with a binary search. Alternately, the discovery that a rehashing function is being called excessively, can lead to a different hash function or a larger hash table. If the data abstraction function cannot easily be speeded up, it may be advantageous to cache its results, and eliminate the need to rerun it for identical inputs. These and other ideas for program improvement are discussed in [Bentley81].

This tool is best used in an iterative approach: profiling the program, eliminating one bottleneck, then finding some other part of the program that begins to dominate execution time. For instance, we have used **gprof** on itself; eliminating, rewriting, and inline expanding routines, until reading data files (hardly a target for optimization!) represents the dominating factor in its execution time.

Certain types of programs are not easily analyzed by **gprof**. They are typified by programs that exhibit a large degree of recursion, such as recursive descent compilers. The problem is that most of the major routines are grouped into a single monolithic cycle. As in the symbol table abstraction that is placed in one routine, it is impossible to distinguish which members of the cycle are responsible for the execution time. Unfortunately there are no easy modifications to these programs that make them amenable to analysis.

A completely different use of the profiler is to analyze the control flow of an unfamiliar program. If you receive a program from another user that you need to modify in some small way, it is often unclear where the changes need to be made. By running the program on an example and then using **gprof**, you can get a view of the structure of the program.

Consider an example in which you need to change the output format of the program. For purposes of this example suppose that the call graph of the output portion of the program has the following structure:



Initially you look through the **gprof** output for the system call ‘‘WRITE’’. The format routine you will need to change is probably among the parents of the ‘‘WRITE’’ procedure. The next step is to look at the profile entry for each of parents of ‘‘WRITE’’, in this example either ‘‘FORMAT1’’ or ‘‘FORMAT2’’, to determine which one to change. Each format routine will have one or more parents, in this example ‘‘CALC1’’, ‘‘CALC2’’, and ‘‘CALC3’’. By inspecting the source code for each of these routines you can determine which format routine generates the output that you wish to modify. Since the **gprof** entry

index	%time	self	descendants	called/total called+self called/total	parents name children	index
		0.20	1.20	4/10	CALLER1	[7]
		0.30	1.80	6/10	CALLER2	[1]

[2]	41.5	0.50	3.00	10+4	EXAMPLE	[2]
		1.50	1.00	20/40	SUB1 <cycle1>	[4]
		0.00	0.50	1/5	SUB2	[9]
		0.00	0.00	0/5	SUB3	[11]

Profile entry for EXAMPLE.
Figure 4.

shows all the potential calls to the format routine you intend to change, you can determine if your modifications will affect output that should be left alone. If you desire to change the output of ''CALC2'', but not ''CALC3'', then formatting routine ''FORMAT2'' needs to be split into two separate routines, one of which implements the new format. You can then retarget just the call by ''CALC2'' that needs the new format. It should be noted that the static call information is particularly useful here since the test case you run probably will not exercise the entire program.

7. Conclusions

We have created a profiler that aids in the evaluation of modular programs. For each routine in the program, the profile shows the extent to which that routine helps support various abstractions, and how that routine uses other abstractions. The profile accurately assesses the cost of routines at all levels of the program decomposition. The profiler is easily used, and can be compiled into the program without any prior planning by the programmer. It adds only five to thirty percent execution overhead to the program being profiled, produces no additional output until after the program finishes, and allows the program to be measured in its actual environment. Finally, the profiler runs on a time-sharing system using only the normal services provided by the operating system and compilers.

8. References

[Bentley81]
Bentley, J. L., ''Writing Efficient Code'', Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, CMU-CS-81-116, 1981.

[Graham82]
Graham, S. L., Henry, R. R., Schulman, R. A., ''An Experiment in Table Driven Code Generation'', SIGPLAN '82 Symposium on Compiler Construction, June, 1982.

[Joy79]
Joy, W. N., Graham, S. L., Haley, C. B. ''Berkeley Pascal User's Manual'', Version 1.1, Computer Science Division University of California, Berkeley, CA. April 1979.

[Knuth71]
Knuth, D. E. ''An empirical study of FORTRAN programs'', Software - Practice and Experience, 1, 105-133. 1971

[Satterthwaite72]
Satterthwaite, E. ''Debugging Tools for High Level Languages'', Software - Practice and Experience, 2, 197-217, 1972

[Tarjan72]
Tarjan, R. E., ''Depth first search and linear graph algorithm,'' *SIAM J. Computing* **1**:2, 146-160, 1972.

[Unix]
Unix Programmer's Manual, ''**prof** command'', section 1, Bell Laboratories, Murray Hill, NJ. January 1979.

Screen Updating and Cursor Movement Optimization: A Library Package

Kenneth C. R. C. Arnold
Elan Amir

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 **termcap**(5) 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, Mary Ann Horton, and Joe Kalash, was, and is, also greatly appreciated. *Ken Arnold 16 April 1986*

The help and/or support of Kirk McKusick and Keith Bostic (public vi!) was invaluable in bringing the package ''into the 90's'', which now includes completely new data structures and screen refresh optimization routines. *Elan Amir 29 December 1992*

1. Overview

In making available the generalized terminal descriptions in **termcap**(5), 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 with nearly as much ease as is necessary to simply print or read things.

1.1. Terminology

In this document, the following terminology is used:

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 Applications

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. Compilations should have the following form:

cc [*flags*] file ... **–lcurses –ltermcap**

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 terminal, in the area covered by the window, look like that window. Note, therefore, that changing something on a window *does not change the terminal*. Actual updates to the terminal screen are made only by calling **refresh**() or **wrefresh**(). This allows the programmer to maintain several different ideas of what a portion of the terminal screen should look like. Also, changes can be made to windows in any order, without regard to motion efficiency. Then, at will, the programmer can effectively say ''make it look like this'', and the package will execute the changes in an optimal way.

1.4. Naming Conventions

As hinted above, the routines can use several windows, but two are always available: *curscr*, which is the image of what the terminal looks like at present, and *stdscr*, which is the image of what the programmer wants the terminal to look like next. The user should not 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 **w**indow-specific **addch**()) is provided[1]. 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 clumsiness, most I/O routines can be preceded by the prefix "**mv**" and the desired (y, x) co-ordinates can then 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 a window pointer is needed, it is always the first parameter 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
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	flag returned by routines upon success.

3. Usage

This is a description of how to actually use the screen package. For simplicity, we assume all updating, reading, etc. is applied to *stdscr*, although a different window can of course be specified.

3.1. Initialization

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 **cbreak**() should be called after **initscr**().

After the initial window allocation done by **initscr**(), specific window characteristics can be set. Scrolling can be enabled by calling **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. Additional windows can be created by using the functions **newwin**() and **subwin**(). **Delwin**() allows you to delete an exisiting window. The variables *LINES* and *COLS* control the size of the terminal. They are initially implicitly set by **initscr**(), but can be altered explicitly by the user followed by a call to **initscr**(). Note that any call to **initscr**(), will always delete any existing *stdscr* and/or *curscr* before creating new ones so this change is best done before the initial call to **initscr**().

3.2. Output

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. As mentioned above, you can combine the two into **mvaddch**() to do both things in one call.

The other output functions (such as **addstr**() and **printw**()) all call **addch**() to add characters to the window.

After a change has been made to the window, you must call **refresh**(). when you want the portion of the terminal covered by the window to reflect the change. 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 routines **touchwin**(), **touchline**(), and **touchoverlap**() are provided to make it look like a desired part of window has been changed, thus forcing **refresh**() to check that whole subsection of the terminal for changes.

If you call **wrefresh**() with *curscr*, it will make the screen look like the image of *curscr*. This is useful for implementing a command which would redraw the screen in case it got messed up.

3.3. 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.

3.4. Termination

In order to perform 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. The terminal state module uses the variable *curses_termios* to save the original terminal state which is then restored upon a call to **endwin**(). Thus, anytime after the call to initscr, **endwin**() should be called before exiting. Note however, that **endwin**() should always be called **before** the final calls to **delwin**(), which free the storage of the windows.

4. Cursor Movement Optimizations

One of the most difficult things to do properly is motion optimization. After using **gettmode**() and **setterm**() to get the terminal descriptions, the function **mvcur**() deals with this task. It usage is

simple: 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, For example, to absolutely address the lower left hand corner of the screen from anywhere just claim that you are in the upper right hand corner:

mvcur(0, COLS–1, LINES–1, 0);

5. Character Output and Scrolling

The character output policy deals with the following problems. First, where is the location of the cursor after a character is printed, and secondly, when does the screen scroll if scrolling is enabled.

In the normal case the characters are output as expected, with the cursor occupying the position of the next character to be output. However, when the cursor is on the last column of the line, the cursor will remain on that position after the last character on the line is output and will only assume the position on the next line when the next character (the first on the next line) is output.

Likewise, if scrolling is enabled, a scroll will be invoked only when the first character on he first line past the bottom line of the window is output. If scrolling is not enabled the chracters will to be output to the bottom right corner of the window which is the cursor location.

This policy allows consistent behavior of the cursor at the boundary conditions. Furthermore, it prevents a scroll from happening before it is actually needed (the old package used to scroll when the bottom right position was output a character). As a precendent, it models the *xterm* character output conventions.

6. Terminal State Handling

The variable *curses_termios* contains the terminal state of the terminal. Certain historical routines return information: **baudrate**(), **erasechar**(), **killchar**(), and **ospeed**(). These routines are obsolete and exist only for backward compatibility. If you wish to use the information in the *curses_termios* structure, you should use the *tsetattr*(3) routines.

7. Subwindows

Subwindows are windows which do not have an independent text structure, *i.e.*, they are windows whose text is a subset of the text of a larger window: the *parent* window. One consequence of this is that changes to either the parent or the child window are destructive to the other, *i.e.*, a change to the subwindow is also a change to the parent window and a change to the parent window in the region defined by the subwindow is implicitly a change to the subwindow as well. Apart from this detail, subwindows function like any other window.

8. The Functions

In the following definitions, "†" means that the "function" is really a "#define" macro with arguments.

addch(*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(*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.

baudrate();†

Returns the output baud rate of the terminal. This is a system dependent constant (defined in **<sys/tty.h>** on BSD systems, which is included by **<curses.h>**).

box(*WINDOW win*, *char vert*, *char 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.

cbreak();†

Set or the terminal to cbreak mode.

clear();†

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(*WINDOW *scr*, *int boolf*);†

Sets the clear flag for the screen *scr*. If *boolf* is non-zero, this will force a clear-screen to be printed on the next **refresh**(), or stop it from doing so if *boolf* is 0. 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();†

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();†

Wipes the window clear from the current (y, x) co-ordinates to the end of the line. This has no associated "**mv**" command.

crmode();†

Identical to **cbreak**(). The misnamed macro **crmode**() and **nocrmode**() is retained for backwards compatibility with ealier versions of the library.

delch();

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();

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.

delwin(*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.

echo();†

Sets the terminal to echo characters.

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 and before the final calls to **delwin**(). It does not exit. This is especially useful for resetting tty stats when trapping rubouts via **signal**(2).

erase();†

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.

erasechar();†

Returns the erase character for the terminal, *i.e.*, the character used by the user to erase a single character from the input.

flushok(*WINDOW *win*, *int boolf*);

Normally, **refresh**() **fflush**(*'s*); *stdout* when it is finished. **flushok**() allows you to control this. if *boolf* is non-zero (*i.e.*, non-zero) it will do the **fflush**(), otherwise it will not.

getch();†

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(*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**()) 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.

gettmode();

Get the tty stats. This is normally called by **initscr**().

getyx(*WINDOW *win*, *int y*, *int 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*.

idlok(*WINDOW *win*, *int boolf*);

Reserved for future use. This will eventually signal to **refresh**() that it is all right to use the insert and delete line sequences when updating the window.

inch();†

Returns the character at the current position on the given window. This does not make any changes to the window.

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* (initially "dumb"). If the boolean *My_term* is non-zero, *Def_term* is always used. If the system supports the **TIOCGWINSZ** *ioctl(2)* call, it is used to get the number of lines and columns for the terminal, otherwise it is taken from the **termcap** description.

insch(*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();

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.

killchar();†

Returns the line kill character for the terminal, *i.e.*, the character used by the user to erase an entire line from the input.

leaveok(*WINDOW *win*, *int boolf*);†

Sets the boolean flag for leaving the cursor after the last change. If *boolf* is non-zero, 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 *boolf*
is 0 the cursor will be moved to the current (y, x) co-ordinates. This flag (initially 0) retains its value until changed by the user.

move(*int y*, *int 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.

mvcur(*int lasty*, *int lastx*, *int newy*, *int 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.

mvprintw(*int y*, *int x*, *const char *fmt*, ...);

Equivalent to:

```
move(y, x);
printw(fmt, ...);
```

mvscanw(*int y*, *int x*, *const char *fmt*, ...);

Equivalent to:

```
move(y, x);
scanw(fmt, ...);
```

mvwin(*WINDOW *win*, *int y*, *int 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. For subwindows, **mvwin**() also returns ERR if you attempt to move it off its main window. If you move a main window, all subwindows are moved along with it.

mvwprintw(*WINDOW *win*, *int y*, *int x*, *const char *fmt*, ...);

Equivalent to:

```
wmove(win, y, x);
printw(fmt, ...);
```

mvwscanw(*WINDOW *win*, *int y*, *int x*, *const char *fmt*, ...);

Equivalent to:

```
wmove(win, y, x);
scanw(fmt, ...);
```

newwin(*int lines*, *int cols*, *int begin_y*, *int 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();†

Set the terminal to 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.

nocbreak();†

Unset the terminal from cbreak mode.

nocrmode();†

Identical to **nocbreak**(). The misnamed macro **nocrmode**() is retained for backwards compatibility with ealier versions of the library.

noecho();†

Turn echoing of characters off.

nonl();†

Unset the terminal to from nl mode. See **nl**().

noraw();†

Unset the terminal from raw mode. See **raw**().

overlay(*WINDOW *win1*, *WINDOW *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. Note that all non-blank characters are overwritten destructively in the overlay.

overwrite(*WINDOW *win1*, *WINDOW *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(*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.

raw();†

Set the terminal to raw mode. On version 7 **UNIX** this also turns off newline mapping (see **nl**()).

refresh();†

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.

As a special case, if **wrefresh**() is called with the window *curscr* the screen is cleared and repainted as it is currently. This is very useful for allowing the redrawing of the screen when the user has garbage dumped on his terminal.

---

[1] Actually, **addch**() is really a "#define" macro with arguments, as are most of the "functions" which act upon *stdscr*.

resetty();†

resetty() restores them to what **savetty**() stored. These functions are performed automatically by **initscr**() and **endwin**(). This function should not be used by the user.

savetty();†

savetty() saves the current tty characteristic flags. See **resetty**(). This function should not be used by the user.

scanw(*char *fmt*, ...);

Perform a **scanf**() through the window using *fmt*. It does this using consecutive calls to **getch**() (or **wgetch**()). This returns ERR if it would cause the screen to scroll illegally.

scroll(*WINDOW *win*);

Scroll the window upward one line. This is normally not used by the user.

scrollok(*WINDOW *win*, *int boolf*);†

Set the scroll flag for the given window. If *boolf* is 0, scrolling is not allowed. This is its default setting.

standend();†

End standout mode initiated by **standout**().

standout();†

Causes any characters added to the window to be put in standout mode on the terminal (if it has that capability).

subwin(*WINDOW *win*, *int lines*, *int cols*, *int begin_y*, *int begin_x*);

Create a new window with *lines* lines and *cols* columns starting at position (*begin_y, begin_x*) inside 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.

touchline(*WINDOW *win*, *int y*, *int startx*, *int endx*);

This function performs a function similar to **touchwin**() on a single line. It marks the first change for the given line to be *startx*, if it is before the current first change mark, and the last change mark is set to be *endx* if it is currently less than *endx*.

touchoverlap(*WINDOW *win1*, *WINDOW *win2*);

Touch the window *win2* in the area which overlaps with *win1*. If they do not overlap, no changes are made.

UNIX is a trademark of Unix System Laboratories.

touchwin(*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.

tstp()

This function will save the current tty state and then put the process to sleep. When the process gets restarted, it restores the saved tty state and then calls **wrefresh**(*curscr*); to redraw the screen. **Initscr**() sets the signal SIGTSTP to trap to this routine.

unctrl(*char *ch*);†

Returns a string which is an ASCII representation of *ch*. Characters are 8 bits long.

unctrllen(*char *ch*);†

Returns the length of the ASCII representation of *ch*.

vwprintw(*WINDOW *win*, *const char *fmt*, *va_list ap*);

Identical to **printw**() except that it takes both a window specification and a pointer to a variable length argument list.

vwscanw(*WINDOW *win*, *const char *fmt*, *va_list ap*);

Identical to **scanw**() except that it takes both a window specification and a pointer to a variable length argument list.

waddbytes(*WINDOW *win*, *char *str*, *int len*);

This function is the low level character output function. *Len* characters of the string *str* are output to the current (y, x) co-ordinates position of the window specified by *win.*

The following functions differ from the standard functions only in their specification of a window, rather than the use of the default stdscr.

waddch(*WINDOW *win*, *char ch*);
waddstr(*WINDOW *win*, *char *str*);
wclear(*WINDOW *win*);
wclrtobot(*WINDOW *win*);
wclrtoeol(*WINDOW *win*);
wdelch(*WINDOW *win*);
wdeleteln(*WINDOW *win*);
werase(*WINDOW *win*);
wgetch(*WINDOW *win*);
wgetstr(*WINDOW *win*, *char *str*);
winch(*WINDOW *win*);†
winsch(*WINDOW *win*, *char c*);
winsertln(*WINDOW *win*);
wmove(*WINDOW *win*, *int y*, *int, x''*);
wprintw(*WINDOW *win*, *char *fmt*, ...);
wrefresh(*WINDOW *win*);
wscanw(*WINDOW *win*, *char *fmt*, ...);

wstandend(*WINDOW *win*);
wstandout(*WINDOW *win*);

1. Examples

Here we present a few examples of how to use the package. They attempt to be representative, though not comprehensive. Further examples can be found in the games section of the source tree and in various utilities that use the screen such as *systat(1)*.

The following examples are intended to demonstrate the basic structure of a program using the package. An additional, more comprehensive, program can be found in the source code in the *examples* subdirectory.

1.1. Simple Character Output

This program demonstrates how to set up a window and output characters to it. Also, it demonstrates how one might control the output to the window. If you run this program, you will get a demonstration of the character output chracteristics discussed in the above Character Output section.

```
#include <sys/types.h>
#include <curses.h>
#include <stdio.h>
#include <signal.h>


#define YSIZE 10
#define XSIZE 20

int quit();

main()
{
        int i, j, c;
        size_t len;
        char id[100];
        FILE *fp;
        char *s;

        initscr();                              /* Always call initscr() first */
        signal(SIGINT, quit);                   /* Make sure wou have a 'cleanup' fn */
        crmode();                               /* We want cbreak mode */
        noecho();                               /* We want to have control of chars */
        delwin(stdscr);                         /* Create our own stdscr */
        stdscr = newwin(YSIZE, XSIZE, 10, 35);
        flushok(stdscr, TRUE);                  /* Enable flushing of stdout */
        scrollok(stdscr, TRUE);                 /* Enable scrolling */
        erase();                                /* Initially, clear the screen */

        standout();
        move(0,0);
        while (1) {
                c = getchar();
                switch(c) {
                case 'q':                       /* Quit on 'q' */
                        quit();
                        break;
```

```
                case ´s´:                       /* Go into standout mode on 's' */
                        standout();
                        break;
                case ´e´:                       /* Exit standout mode on 'e' */
                        standend();
                        break;
                case ´r´:                       /* Force a refresh on 'r' */
                        wrefresh(curscr);
                        break;
                default:                        /* By default output the character */
                        addch(c);
                        refresh();
                }
        }
}


int
quit()
{
        erase();                        /* Terminate by erasing the screen */
        refresh();
        endwin();                       /* Always end with endwin() */
        delwin(curscr);                 /* Return storage */
        delwin(stdscr);
        putchar(´\n´);
        exit(0);
}
```

1.2. Twinkle

This is a moderately simple program which prints patterns on the screen. 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

typedef struct {
        int     y, x;
} LOCS;
```

```
LOCS    Layout[NCOLS * NLINES];     /* current board layout */

int     Pattern,                    /* current pattern number */
        Numstars;                   /* number of stars in pattern */

char    *getenv();

int     die();

main()
{
        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;
                                    lp++;
                            }
            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();
```

```
            }
    }
```

Contents

1 Overview 3
1.1 Terminology 3
1.2 Compiling Applications 3
1.3 Screen Updating 3
1.4 Naming Conventions 4
2 Variables 4
3 Usage 4
3.1 Initialization 4
3.2 Output 5
3.3 Input 5
3.4 Termination 5
4 Cursor Movement Optimizations 5
5 Character Output and Scrolling 6
6 Terminal State Handling 6
7 Subwindows 6
8 The Functions 6
Appendix A 15
1 Examples 15
1.1 Simple Character Output 15
1.2 Twinkle 16

An Introductory 4.4BSD Interprocess Communication Tutorial

Stuart Sechrest

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

ABSTRACT

Berkeley UNIX† 4.4BSD offers several choices for interprocess communication. To aid the programmer in developing programs which are comprised of cooperating processes, the different choices are discussed and a series of example programs are presented. These programs demonstrate in a simple way the use of pipes, socketpairs, sockets and the use of datagram and stream communication. The intent of this document is to present a few simple example programs, not to describe the networking system in full.

1. Goals

Facilities for interprocess communication (IPC) and networking were a major addition to UNIX in the Berkeley UNIX 4.2BSD release. These facilities required major additions and some changes to the system interface. The basic idea of this interface is to make IPC similar to file I/O. In UNIX a process has a set of I/O descriptors, from which one reads and to which one writes. Descriptors may refer to normal files, to devices (including terminals), or to communication channels. The use of a descriptor has three phases: its creation, its use for reading and writing, and its destruction. By using descriptors to write files, rather than simply naming the target file in the write call, one gains a surprising amount of flexibility. Often, the program that creates a descriptor will be different from the program that uses the descriptor. For example the shell can create a descriptor for the output of the 'ls' command that will cause the listing to appear in a file rather than on a terminal. Pipes are another form of descriptor that have been used in UNIX for some time. Pipes allow one-way data transmission from one process to another; the two processes and the pipe must be set up by a common ancestor.

The use of descriptors is not the only communication interface provided by UNIX. The signal mechanism sends a tiny amount of information from one process to another. The signaled process receives only the signal type, not the identity of the sender, and the number of possible signals is small. The signal semantics limit the flexibility of the signaling mechanism as a means of interprocess communication.

The identification of IPC with I/O is quite longstanding in UNIX and has proved quite successful. At first, however, IPC was limited to processes communicating within a single machine. With Berkeley UNIX 4.2BSD this expanded to include IPC between machines. This expansion has necessitated some change in the way that descriptors are created. Additionally, new possibilities for the meaning of read and write have been admitted. Originally the meanings, or semantics, of these terms were fairly simple. When you wrote something it was delivered. When you read something, you were blocked until the data arrived. Other possibilities exist, however. One can write without full assurance of delivery if one

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

can check later to catch occasional failures. Messages can be kept as discrete units or merged into a stream. One can ask to read, but insist on not waiting if nothing is immediately available. These new possibilities are allowed in the Berkeley UNIX IPC interface.

Thus Berkeley UNIX 4.4BSD offers several choices for IPC. This paper presents simple examples that illustrate some of the choices. The reader is presumed to be familiar with the C programming language [Kernighan & Ritchie 1978], but not necessarily with the system calls of the UNIX system or with processes and interprocess communication. The paper reviews the notion of a process and the types of communication that are supported by Berkeley UNIX 4.4BSD. A series of examples are presented that create processes that communicate with one another. The programs show different ways of establishing channels of communication. Finally, the calls that actually transfer data are reviewed. To clearly present how communication can take place, the example programs have been cleared of anything that might be construed as useful work. They can, therefore, serve as models for the programmer trying to construct programs which are comprised of cooperating processes.

2. Processes

A *program* is both a sequence of statements and a rough way of referring to the computation that occurs when the compiled statements are run. A *process* can be thought of as a single line of control in a program. Most programs execute some statements, go through a few loops, branch in various directions and then end. These are single process programs. Programs can also have a point where control splits into two independent lines, an action called *forking.* In UNIX these lines can never join again. A call to the system routine *fork()*, causes a process to split in this way. The result of this call is that two independent processes will be running, executing exactly the same code. Memory values will be the same for all values set before the fork, but, subsequently, each version will be able to change only the value of its own copy of each variable. Initially, the only difference between the two will be the value returned by *fork().* The parent will receive a process id for the child, the child will receive a zero. Calls to *fork(),* therefore, typically precede, or are included in, an if-statement.

A process views the rest of the system through a private table of descriptors. The descriptors can represent open files or sockets (sockets are communication objects that will be discussed below). Descriptors are referred to by their index numbers in the table. The first three descriptors are often known by special names, *stdin, stdout* and *stderr*. These are the standard input, output and error. When a process forks, its descriptor table is copied to the child. Thus, if the parent's standard input is being taken from a terminal (devices are also treated as files in UNIX), the child's input will be taken from the same terminal. Whoever reads first will get the input. If, before forking, the parent changes its standard input so that it is reading from a new file, the child will take its input from the new file. It is also possible to take input from a socket, rather than from a file.

3. Pipes

Most users of UNIX know that they can pipe the output of a program ‘‘prog1’’ to the input of another, ‘‘prog2,’’ by typing the command *“prog1 | prog2.”* This is called ‘‘piping’’ the output of one program to another because the mechanism used to transfer the output is called a pipe. When the user types a command, the command is read by the shell, which decides how to execute it. If the command is simple, for example, *“prog1,”* the shell forks a process, which executes the program, prog1, and then dies. The shell waits for this termination and then prompts for the next command. If the command is a compound command, *“prog1 | prog2,”* the shell creates two processes connected by a pipe. One process runs the program, prog1, the other runs prog2. The pipe is an I/O mechanism with two ends, or sockets. Data that is written into one socket can be read from the other.

Since a program specifies its input and output only by the descriptor table indices, which appear as variables or constants, the input source and output destination can be changed without changing the text of the program. It is in this way that the shell is able to set up pipes. Before executing prog1, the process can close whatever is at *stdout* and replace it with one end of a pipe. Similarly, the process that will execute prog2 can substitute the opposite end of the pipe for *stdin.*

Let us now examine a program that creates a pipe for communication between its child and itself (see Figure 1; all figures appear at the end of the document). A pipe is created by a parent process, which then forks. When a process forks, the parent's descriptor table is copied into the child's.

In Figure 1, the parent process makes a call to the system routine *pipe()*. This routine creates a pipe and places descriptors for the sockets for the two ends of the pipe in the process's descriptor table. *Pipe()* is passed an array into which it places the index numbers of the sockets it created. The two ends are not equivalent. The socket whose index is returned in the low word of the array is opened for reading only, while the socket in the high end is opened only for writing. This corresponds to the fact that the standard input is the first descriptor of a process's descriptor table and the standard output is the second. After creating the pipe, the parent creates the child with which it will share the pipe by calling *fork()*. Figure 2 illustrates the effect of a fork. The parent process's descriptor table points to both ends of the pipe. After the fork, both parent's and child's descriptor tables point to the pipe. The child can then use the pipe to send a message to the parent.

Just what is a pipe? It is a one-way communication mechanism, with one end opened for reading and the other end for writing. Therefore, parent and child need to agree on which way to turn the pipe, from parent to child or the other way around. Using the same pipe for communication both from parent to child and from child to parent would be possible (since both processes have references to both ends), but very complicated. If the parent and child are to have a two-way conversation, the parent creates two pipes, one for use in each direction. (In accordance with their plans, both parent and child in the example above close the socket that they will not use. It is not required that unused descriptors be closed, but it is good practice.) A pipe is also a *stream* communication mechanism; that is, all messages sent through the pipe are placed in order and reliably delivered. When the reader asks for a certain number of bytes from this stream, he is given as many bytes as are available, up to the amount of the request. Note that these bytes may have come from the same call to *write()* or from several calls to *write()* which were concatenated.

4. Socketpairs

Berkeley UNIX 4.4BSD provides a slight generalization of pipes. A pipe is a pair of connected sockets for one-way stream communication. One may obtain a pair of connected sockets for two-way stream communication by calling the routine *socketpair()*. The program in Figure 3 calls *socketpair()* to create such a connection. The program uses the link for communication in both directions. Since socketpairs are an extension of pipes, their use resembles that of pipes. Figure 4 illustrates the result of a fork following a call to *socketpair()*.

Socketpair() takes as arguments a specification of a domain, a style of communication, and a protocol. These are the parameters shown in the example. Domains and protocols will be discussed in the next section. Briefly, a domain is a space of names that may be bound to sockets and implies certain other conventions. Currently, socketpairs have only been implemented for one domain, called the UNIX domain. The UNIX domain uses UNIX path names for naming sockets. It only allows communication between sockets on the same machine.

Note that the header files *<sys/socket.h>* and *<sys/types.h>*. are required in this program. The constants AF_UNIX and SOCK_STREAM are defined in *<sys/socket.h>*, which in turn requires the file *<sys/types.h>* for some of its definitions.

5. Domains and Protocols

Pipes and socketpairs are a simple solution for communicating between a parent and child or between child processes. What if we wanted to have processes that have no common ancestor with whom to set up communication? Neither standard UNIX pipes nor socketpairs are the answer here, since both mechanisms require a common ancestor to set up the communication. We would like to have two processes separately create sockets and then have messages sent between them. This is often the case when providing or using a service in the system. This is also the case when the communicating processes are on separate machines. In Berkeley UNIX 4.4BSD one can create individual sockets, give

them names and send messages between them.

Sockets created by different programs use names to refer to one another; names generally must be translated into addresses for use. The space from which an address is drawn is referred to as a *domain.* There are several domains for sockets. Two that will be used in the examples here are the UNIX domain (or AF_UNIX, for Address Format UNIX) and the Internet domain (or AF_INET). UNIX domain IPC is an experimental facility in 4.2BSD and 4.3BSD. In the UNIX domain, a socket is given a path name within the file system name space. A file system node is created for the socket and other processes may then refer to the socket by giving the proper pathname. UNIX domain names, therefore, allow communication between any two processes that work in the same file system. The Internet domain is the UNIX implementation of the DARPA Internet standard protocols IP/TCP/UDP. Addresses in the Internet domain consist of a machine network address and an identifying number, called a port. Internet domain names allow communication between machines.

Communication follows some particular "style." Currently, communication is either through a *stream* or by *datagram.* Stream communication implies several things. Communication takes place across a connection between two sockets. The communication is reliable, error-free, and, as in pipes, no message boundaries are kept. Reading from a stream may result in reading the data sent from one or several calls to *write()* or only part of the data from a single call, if there is not enough room for the entire message, or if not all the data from a large message has been transferred. The protocol implementing such a style will retransmit messages received with errors. It will also return error messages if one tries to send a message after the connection has been broken. Datagram communication does not use connections. Each message is addressed individually. If the address is correct, it will generally be received, although this is not guaranteed. Often datagrams are used for requests that require a response from the recipient. If no response arrives in a reasonable amount of time, the request is repeated. The individual datagrams will be kept separate when they are read, that is, message boundaries are preserved.

The difference in performance between the two styles of communication is generally less important than the difference in semantics. The performance gain that one might find in using datagrams must be weighed against the increased complexity of the program, which must now concern itself with lost or out of order messages. If lost messages may simply be ignored, the quantity of traffic may be a consideration. The expense of setting up a connection is best justified by frequent use of the connection. Since the performance of a protocol changes as it is tuned for different situations, it is best to seek the most up-to-date information when making choices for a program in which performance is crucial.

A protocol is a set of rules, data formats and conventions that regulate the transfer of data between participants in the communication. In general, there is one protocol for each socket type (stream, datagram, etc.) within each domain. The code that implements a protocol keeps track of the names that are bound to sockets, sets up connections and transfers data between sockets, perhaps sending the data across a network. This code also keeps track of the names that are bound to sockets. It is possible for several protocols, differing only in low level details, to implement the same style of communication within a particular domain. Although it is possible to select which protocol should be used, for nearly all uses it is sufficient to request the default protocol. This has been done in all of the example programs.

One specifies the domain, style and protocol of a socket when it is created. For example, in Figure 5a the call to *socket()* causes the creation of a datagram socket with the default protocol in the UNIX domain.

6. Datagrams in the UNIX Domain

Let us now look at two programs that create sockets separately. The programs in Figures 5a and 5b use datagram communication rather than a stream. The structure used to name UNIX domain sockets is defined in the file *<sys/un.h>*. The definition has also been included in the example for clarity.

Each program creates a socket with a call to *socket().* These sockets are in the UNIX domain. Once a name has been decided upon it is attached to a socket by the system call *bind().* The program

in Figure 5a uses the name ''socket'', which it binds to its socket. This name will appear in the working directory of the program. The routines in Figure 5b use its socket only for sending messages. It does not create a name for the socket because no other process has to refer to it.

Names in the UNIX domain are path names. Like file path names they may be either absolute (e.g. ''/dev/imaginary'') or relative (e.g. ''socket''). Because these names are used to allow processes to rendezvous, relative path names can pose difficulties and should be used with care. When a name is bound into the name space, a file (inode) is allocated in the file system. If the inode is not deallocated, the name will continue to exist even after the bound socket is closed. This can cause subsequent runs of a program to find that a name is unavailable, and can cause directories to fill up with these objects. The names are removed by calling *unlink()* or using the *rm* (1) command. Names in the UNIX domain are only used for rendezvous. They are not used for message delivery once a connection is established. Therefore, in contrast with the Internet domain, unbound sockets need not be (and are not) automatically given addresses when they are connected.

There is no established means of communicating names to interested parties. In the example, the program in Figure 5b gets the name of the socket to which it will send its message through its command line arguments. Once a line of communication has been created, one can send the names of additional, perhaps new, sockets over the link. Facilities will have to be built that will make the distribution of names less of a problem than it now is.

7. Datagrams in the Internet Domain

The examples in Figure 6a and 6b are very close to the previous example except that the socket is in the Internet domain. The structure of Internet domain addresses is defined in the file <*netinet/in.h*>. Internet addresses specify a host address (a 32-bit number) and a delivery slot, or port, on that machine. These ports are managed by the system routines that implement a particular protocol. Unlike UNIX domain names, Internet socket names are not entered into the file system and, therefore, they do not have to be unlinked after the socket has been closed. When a message must be sent between machines it is sent to the protocol routine on the destination machine, which interprets the address to determine to which socket the message should be delivered. Several different protocols may be active on the same machine, but, in general, they will not communicate with one another. As a result, different protocols are allowed to use the same port numbers. Thus, implicitly, an Internet address is a triple including a protocol as well as the port and machine address. An *association* is a temporary or permanent specification of a pair of communicating sockets. An association is thus identified by the tuple <*protocol, local machine address, local port, remote machine address, remote port*>. An association may be transient when using datagram sockets; the association actually exists during a *send* operation.

The protocol for a socket is chosen when the socket is created. The local machine address for a socket can be any valid network address of the machine, if it has more than one, or it can be the wildcard value INADDR_ANY. The wildcard value is used in the program in Figure 6a. If a machine has several network addresses, it is likely that messages sent to any of the addresses should be deliverable to a socket. This will be the case if the wildcard value has been chosen. Note that even if the wildcard value is chosen, a program sending messages to the named socket must specify a valid network address. One can be willing to receive from ''anywhere,'' but one cannot send a message ''anywhere.'' The program in Figure 6b is given the destination host name as a command line argument. To determine a network address to which it can send the message, it looks up the host address by the call to *gethostbyname()*. The returned structure includes the host's network address, which is copied into the structure specifying the destination of the message.

The port number can be thought of as the number of a mailbox, into which the protocol places one's messages. Certain daemons, offering certain advertised services, have reserved or ''well-known'' port numbers. These fall in the range from 1 to 1023. Higher numbers are available to general users. Only servers need to ask for a particular number. The system will assign an unused port number when an address is bound to a socket. This may happen when an explicit *bind* call is made with a port number of 0, or when a *connect* or *send* is performed on an unbound socket. Note that port numbers

are not automatically reported back to the user. After calling *bind(),* asking for port 0, one may call *getsockname()* to discover what port was actually assigned. The routine *getsockname()* will not work for names in the UNIX domain.

The format of the socket address is specified in part by standards within the Internet domain. The specification includes the order of the bytes in the address. Because machines differ in the internal representation they ordinarily use to represent integers, printing out the port number as returned by *getsockname()* may result in a misinterpretation. To print out the number, it is necessary to use the routine *ntohs()* (for *network to host: short*) to convert the number from the network representation to the host's representation. On some machines, such as 68000-based machines, this is a null operation. On others, such as VAXes, this results in a swapping of bytes. Another routine exists to convert a short integer from the host format to the network format, called *htons()*; similar routines exist for long integers. For further information, refer to the entry for *byteorder* in section 3 of the manual.

8. Connections

To send data between stream sockets (having communication style SOCK_STREAM), the sockets must be connected. Figures 7a and 7b show two programs that create such a connection. The program in 7a is relatively simple. To initiate a connection, this program simply creates a stream socket, then calls *connect()*, specifying the address of the socket to which it wishes its socket connected. Provided that the target socket exists and is prepared to handle a connection, connection will be complete, and the program can begin to send messages. Messages will be delivered in order without message boundaries, as with pipes. The connection is destroyed when either socket is closed (or soon thereafter). If a process persists in sending messages after the connection is closed, a SIGPIPE signal is sent to the process by the operating system. Unless explicit action is taken to handle the signal (see the manual page for *signal* or *sigvec*), the process will terminate and the shell will print the message "broken pipe."

Forming a connection is asymmetrical; one process, such as the program in Figure 7a, requests a connection with a particular socket, the other process accepts connection requests. Before a connection can be accepted a socket must be created and an address bound to it. This situation is illustrated in the top half of Figure 8. Process 2 has created a socket and bound a port number to it. Process 1 has created an unnamed socket. The address bound to process 2's socket is then made known to process 1 and, perhaps to several other potential communicants as well. If there are several possible communicants, this one socket might receive several requests for connections. As a result, a new socket is created for each connection. This new socket is the endpoint for communication within this process for this connection. A connection may be destroyed by closing the corresponding socket.

The program in Figure 7b is a rather trivial example of a server. It creates a socket to which it binds a name, which it then advertises. (In this case it prints out the socket number.) The program then calls *listen()* for this socket. Since several clients may attempt to connect more or less simultaneously, a queue of pending connections is maintained in the system address space. *Listen()* marks the socket as willing to accept connections and initializes the queue. When a connection is requested, it is listed in the queue. If the queue is full, an error status may be returned to the requester. The maximum length of this queue is specified by the second argument of *listen()*; the maximum length is limited by the system. Once the listen call has been completed, the program enters an infinite loop. On each pass through the loop, a new connection is accepted and removed from the queue, and, hence, a new socket for the connection is created. The bottom half of Figure 8 shows the result of Process 1 connecting with the named socket of Process 2, and Process 2 accepting the connection. After the connection is created, the service, in this case printing out the messages, is performed and the connection socket closed. The *accept()* call will take a pending connection request from the queue if one is available, or block waiting for a request. Messages are read from the connection socket. Reads from an active connection will normally block until data is available. The number of bytes read is returned. When a connection is destroyed, the read call returns immediately. The number of bytes returned will be zero.

The program in Figure 7c is a slight variation on the server in Figure 7b. It avoids blocking when there are no pending connection requests by calling *select()* to check for pending requests before calling

accept(). This strategy is useful when connections may be received on more than one socket, or when data may arrive on other connected sockets before another connection request.

The programs in Figures 9a and 9b show a program using stream communication in the UNIX domain. Streams in the UNIX domain can be used for this sort of program in exactly the same way as Internet domain streams, except for the form of the names and the restriction of the connections to a single file system. There are some differences, however, in the functionality of streams in the two domains, notably in the handling of *out-of-band* data (discussed briefly below). These differences are beyond the scope of this paper.

9. Reads, Writes, Recvs, etc.

UNIX 4.4BSD has several system calls for reading and writing information. The simplest calls are *read()* and *write()*. *Write()* takes as arguments the index of a descriptor, a pointer to a buffer containing the data and the size of the data. The descriptor may indicate either a file or a connected socket. "Connected" can mean either a connected stream socket (as described in Section 8) or a datagram socket for which a *connect()* call has provided a default destination (see the *connect()* manual page). *Read()* also takes a descriptor that indicates either a file or a socket. *Write()* requires a connected socket since no destination is specified in the parameters of the system call. *Read()* can be used for either a connected or an unconnected socket. These calls are, therefore, quite flexible and may be used to write applications that require no assumptions about the source of their input or the destination of their output. There are variations on *read()* and *write()* that allow the source and destination of the input and output to use several separate buffers, while retaining the flexibility to handle both files and sockets. These are *readv()* and *writev()*, for read and write *vector*.

It is sometimes necessary to send high priority data over a connection that may have unread low priority data at the other end. For example, a user interface process may be interpreting commands and sending them on to another process through a stream connection. The user interface may have filled the stream with as yet unprocessed requests when the user types a command to cancel all outstanding requests. Rather than have the high priority data wait to be processed after the low priority data, it is possible to send it as *out-of-band* (OOB) data. The notification of pending OOB data results in the generation of a SIGURG signal, if this signal has been enabled (see the manual page for *signal* or *sigvec*). See [Leffler 1986] for a more complete description of the OOB mechanism. There are a pair of calls similar to *read* and *write* that allow options, including sending and receiving OOB information; these are *send()* and *recv()*. These calls are used only with sockets; specifying a descriptor for a file will result in the return of an error status. These calls also allow *peeking* at data in a stream. That is, they allow a process to read data without removing the data from the stream. One use of this facility is to read ahead in a stream to determine the size of the next item to be read. When not using these options, these calls have the same functions as *read()* and *write()*.

To send datagrams, one must be allowed to specify the destination. The call *sendto()* takes a destination address as an argument and is therefore used for sending datagrams. The call *recvfrom()* is often used to read datagrams, since this call returns the address of the sender, if it is available, along with the data. If the identity of the sender does not matter, one may use *read()* or *recv()*.

Finally, there are a pair of calls that allow the sending and receiving of messages from multiple buffers, when the address of the recipient must be specified. These are *sendmsg()* and *recvmsg()*. These calls are actually quite general and have other uses, including, in the UNIX domain, the transmission of a file descriptor from one process to another.

The various options for reading and writing are shown in Figure 10, together with their parameters. The parameters for each system call reflect the differences in function of the different calls. In the examples given in this paper, the calls *read()* and *write()* have been used whenever possible.

10. Choices

This paper has presented examples of some of the forms of communication supported by Berkeley UNIX 4.4BSD. These have been presented in an order chosen for ease of presentation. It is useful to

review these options emphasizing the factors that make each attractive.

Pipes have the advantage of portability, in that they are supported in all UNIX systems. They also are relatively simple to use. Socketpairs share this simplicity and have the additional advantage of allowing bidirectional communication. The major shortcoming of these mechanisms is that they require communicating processes to be descendants of a common process. They do not allow intermachine communication.

The two communication domains, UNIX and Internet, allow processes with no common ancestor to communicate. Of the two, only the Internet domain allows communication between machines. This makes the Internet domain a necessary choice for processes running on separate machines.

The choice between datagrams and stream communication is best made by carefully considering the semantic and performance requirements of the application. Streams can be both advantageous and disadvantageous. One disadvantage is that a process is only allowed a limited number of open streams, as there are usually only 64 entries available in the open descriptor table. This can cause problems if a single server must talk with a large number of clients. Another is that for delivering a short message the stream setup and teardown time can be unnecessarily long. Weighed against this are the reliability built into the streams. This will often be the deciding factor in favor of streams.

11. What to do Next

Many of the examples presented here can serve as models for multiprocess programs and for programs distributed across several machines. In developing a new multiprocess program, it is often easiest to first write the code to create the processes and communication paths. After this code is debugged, the code specific to the application can be added.

An introduction to the UNIX system and programming using UNIX system calls can be found in [Kernighan and Pike 1984]. Further documentation of the Berkeley UNIX 4.4BSD IPC mechanisms can be found in [Leffler et al. 1986]. More detailed information about particular calls and protocols is provided in sections 2, 3 and 4 of the UNIX Programmer's Manual [CSRG 1986]. In particular the following manual pages are relevant:

creating and naming sockets	socket(2), bind(2)
establishing connections	listen(2), accept(2), connect(2)
transferring data	read(2), write(2), send(2), recv(2)
addresses	inet(4F)
protocols	tcp(4P), udp(4P).

Acknowledgements

I would like to thank Sam Leffler and Mike Karels for their help in understanding the IPC mechanisms and all the people whose comments have helped in writing and improving this report.

This work was sponsored by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 4031, monitored by the Naval Electronics Systems Command under contract No. N00039-C-0235. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the US Government.

References

B.W. Kernighan & R. Pike, 1984,
The UNIX Programming Environment.
Englewood Cliffs, N.J.: Prentice-Hall.

B.W. Kernighan & D.M. Ritchie, 1978,
The C Programming Language,
Englewood Cliffs, N.J.: Prentice-Hall.

S.J. Leffler, R.S. Fabry, W.N. Joy, P. Lapsley, S. Miller & C. Torek, 1986,
An Advanced 4.4BSD Interprocess Communication Tutorial.
Computer Systems Research Group,
Department of Electrical Engineering and Computer Science,
University of California, Berkeley.

Computer Systems Research Group, 1986,
UNIX Programmer's Manual, 4.4 Berkeley Software Distribution.
Computer Systems Research Group,
Department of Electrical Engineering and Computer Science,
University of California, Berkeley.

```
#define DATA "Bright star, would I were steadfast as thou art . . ."

/*
 * This program creates a pipe, then forks.  The child communicates to the
 * parent over the pipe. Notice that a pipe is a one-way communications
 * device.  I can write to the output socket (sockets[1], the second socket
 * of the array returned by pipe()) and read from the input socket
 * (sockets[0]), but not vice versa.
 */

main()
{
      int sockets[2], child;

      /* Create a pipe */
      if (pipe(sockets) < 0) {
            perror("opening stream socket pair");
            exit(10);
      }

      if ((child = fork()) == -1)
            perror("fork");
      else if (child) {
            char buf[1024];

            /* This is still the parent.  It reads the child's message. */
            close(sockets[1]);
            if (read(sockets[0], buf, 1024) < 0)
                  perror("reading message");
            printf("-->%s\n", buf);
            close(sockets[0]);
      } else {
            /* This is the child.  It writes a message to its parent. */
            close(sockets[0]);
            if (write(sockets[1], DATA, sizeof(DATA)) < 0)
                  perror("writing message");
            close(sockets[1]);
      }
}
```

Figure 1 Use of a pipe

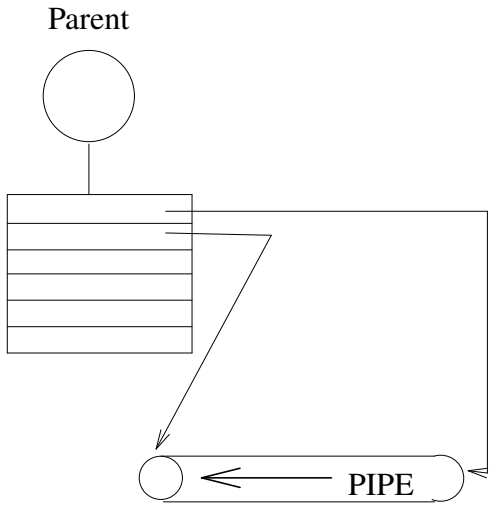


Figure 2 Sharing a pipe between parent and child

```
#include <sys/socket.h>
#include <stdio.h>

#define DATA1 "In Xanadu, did Kublai Khan . . ."
#define DATA2 "A stately pleasure dome decree . . ."

/*
 * This program creates a pair of connected sockets then forks and
 * communicates over them.  This is very similar to communication with pipes,
 * however, socketpairs are two-way communications objects. Therefore I can
 * send messages in both directions.
 */

main()
{
      int sockets[2], child;
      char buf[1024];

      if (socketpair(AF_UNIX, SOCK_STREAM, 0, sockets) < 0) {
            perror("opening stream socket pair");
            exit(1);
      }

      if ((child = fork()) == -1)
            perror("fork");
      else if (child) { /* This is the parent. */
            close(sockets[0]);
            if (read(sockets[1], buf, 1024, 0) < 0)
                  perror("reading stream message");
            printf("-->%s\n", buf);
            if (write(sockets[1], DATA2, sizeof(DATA2)) < 0)
                  perror("writing stream message");
            close(sockets[1]);
      } else {          /* This is the child. */
            close(sockets[1]);
            if (write(sockets[0], DATA1, sizeof(DATA1)) < 0)
                  perror("writing stream message");
            if (read(sockets[0], buf, 1024, 0) < 0)
                  perror("reading stream message");
            printf("-->%s\n", buf);
            close(sockets[0]);
      }
}
```

Figure 3 Use of a socketpair

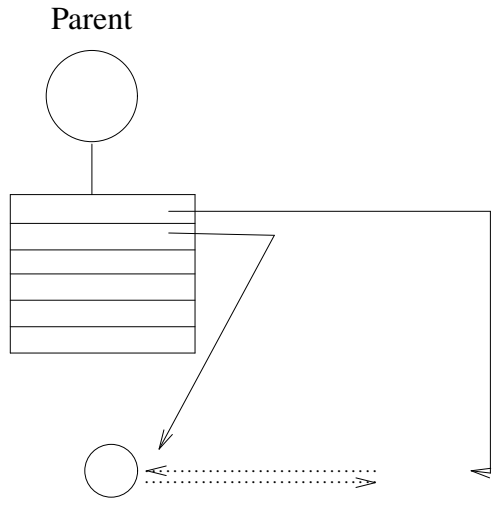


Figure 4 Sharing a socketpair between parent and child

```
#include <sys/socket.h>
#include <sys/un.h>

/*
 * In the included file <sys/un.h> a sockaddr_un is defined as follows
 * struct sockaddr_un {
 *    short sun_family;
 *    char  sun_path[108];
 * };
 */

#include <stdio.h>

#define NAME "socket"

/*
 * This program creates a UNIX domain datagram socket, binds a name to it,
 * then reads from the socket.
 */
main()
{
      int sock, length;
      struct sockaddr_un name;
      char buf[1024];

      /* Create socket from which to read. */
      sock = socket(AF_UNIX, SOCK_DGRAM, 0);
      if (sock < 0) {
            perror("opening datagram socket");
            exit(1);
      }
      /* Create name. */
      name.sun_family = AF_UNIX;
      strcpy(name.sun_path, NAME);
      if (bind(sock, &name, sizeof(struct sockaddr_un))) {
            perror("binding name to datagram socket");
            exit(1);
      }
      printf("socket -->%s\n", NAME);
      /* Read from the socket */
      if (read(sock, buf, 1024) < 0)
            perror("receiving datagram packet");
      printf("-->%s\n", buf);
      close(sock);
      unlink(NAME);
}
```

Figure 5a Reading UNIX domain datagrams

```
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>
```

```
#define DATA "The sea is calm tonight, the tide is full . . ."

/*
 * Here I send a datagram to a receiver whose name I get from the command
 * line arguments.  The form of the command line is udgramsend pathname
 */

main(argc, argv)
      int argc;
      char *argv[];
{
      int sock;
      struct sockaddr_un name;

      /* Create socket on which to send. */
      sock = socket(AF_UNIX, SOCK_DGRAM, 0);
      if (sock < 0) {
            perror("opening datagram socket");
            exit(1);
      }
      /* Construct name of socket to send to. */
      name.sun_family = AF_UNIX;
      strcpy(name.sun_path, argv[1]);
      /* Send message. */
      if (sendto(sock, DATA, sizeof(DATA), 0,
          &name, sizeof(struct sockaddr_un)) < 0) {
            perror("sending datagram message");
      }
      close(sock);
}
```

Figure 5b Sending a UNIX domain datagrams

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>

/*
 * In the included file <netinet/in.h> a sockaddr_in is defined as follows:
 * struct sockaddr_in {
 *    short sin_family;
 *    u_short     sin_port;
 *    struct in_addr sin_addr;
 *    char  sin_zero[8];
 * };
 *
 * This program creates a datagram socket, binds a name to it, then reads
 * from the socket.
 */
main()
{
      int sock, length;
      struct sockaddr_in name;
      char buf[1024];
```

```
	/* Create socket from which to read. */
	sock = socket(AF_INET, SOCK_DGRAM, 0);
	if (sock < 0) {
		perror("opening datagram socket");
		exit(1);
	}
	/* Create name with wildcards. */
	name.sin_family = AF_INET;
	name.sin_addr.s_addr = INADDR_ANY;
	name.sin_port = 0;
	if (bind(sock, &name, sizeof(name))) {
		perror("binding datagram socket");
		exit(1);
	}
	/* Find assigned port value and print it out. */
	length = sizeof(name);
	if (getsockname(sock, &name, &length)) {
		perror("getting socket name");
		exit(1);
	}
	printf("Socket has port #%d\n", ntohs(name.sin_port));
	/* Read from the socket */
	if (read(sock, buf, 1024) < 0)
		perror("receiving datagram packet");
	printf("-->%s\n", buf);
	close(sock);
}
```

Figure 6a Reading Internet domain datagrams

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define DATA "The sea is calm tonight, the tide is full . . ."

/*
 * Here I send a datagram to a receiver whose name I get from the command
 * line arguments.  The form of the command line is dgramsend hostname
 * portnumber
 */

main(argc, argv)
	int argc;
	char *argv[];
{
	int sock;
	struct sockaddr_in name;
	struct hostent *hp, *gethostbyname();

	/* Create socket on which to send. */
	sock = socket(AF_INET, SOCK_DGRAM, 0);
	if (sock < 0) {
```

```
            perror("opening datagram socket");
            exit(1);
      }
      /*
       * Construct name, with no wildcards, of the socket to send to.
       * Getnostbyname() returns a structure including the network address
       * of the specified host.  The port number is taken from the command
       * line.
       */
      hp = gethostbyname(argv[1]);
      if (hp == 0) {
            fprintf(stderr, "%s: unknown host0, argv[1]);
            exit(2);
      }
      bcopy(hp->h_addr, &name.sin_addr, hp->h_length);
      name.sin_family = AF_INET;
      name.sin_port = htons(atoi(argv[2]));
      /* Send message. */
      if (sendto(sock, DATA, sizeof(DATA), 0, &name, sizeof(name)) < 0)
            perror("sending datagram message");
      close(sock);
}
```

Figure 6b Sending an Internet domain datagram

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define DATA "Half a league, half a league . . ."

/*
 * This program creates a socket and initiates a connection with the socket
 * given in the command line.  One message is sent over the connection and
 * then the socket is closed, ending the connection. The form of the command
 * line is streamwrite hostname portnumber
 */

main(argc, argv)
      int argc;
      char *argv[];
{
      int sock;
      struct sockaddr_in server;
      struct hostent *hp, *gethostbyname();
      char buf[1024];

      /* Create socket */
      sock = socket(AF_INET, SOCK_STREAM, 0);
      if (sock < 0) {
            perror("opening stream socket");
            exit(1);
      }
```

```
        /* Connect socket using name specified by command line. */
        server.sin_family = AF_INET;
        hp = gethostbyname(argv[1]);
        if (hp == 0) {
                fprintf(stderr, "%s: unknown host0, argv[1]);
                exit(2);
        }
        bcopy(hp->h_addr, &server.sin_addr, hp->h_length);
        server.sin_port = htons(atoi(argv[2]));

        if (connect(sock, &server, sizeof(server)) < 0) {
                perror("connecting stream socket");
                exit(1);
        }
        if (write(sock, DATA, sizeof(DATA)) < 0)
                perror("writing on stream socket");
        close(sock);
}
```

Figure 7a Initiating an Internet domain stream connection

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1

/*
 * This program creates a socket and then begins an infinite loop. Each time
 * through the loop it accepts a connection and prints out messages from it.
 * When the connection breaks, or a termination message comes through, the
 * program accepts a new connection.
 */

main()
{
        int sock, length;
        struct sockaddr_in server;
        int msgsock;
        char buf[1024];
        int rval;
        int i;

        /* Create socket */
        sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0) {
                perror("opening stream socket");
                exit(1);
        }
        /* Name socket using wildcards */
        server.sin_family = AF_INET;
        server.sin_addr.s_addr = INADDR_ANY;
        server.sin_port = 0;
        if (bind(sock, &server, sizeof(server))) {
```

```
            perror("binding stream socket");
            exit(1);
      }
      /* Find out assigned port number and print it out */
      length = sizeof(server);
      if (getsockname(sock, &server, &length)) {
            perror("getting socket name");
            exit(1);
      }
      printf("Socket has port #%d\n", ntohs(server.sin_port));

      /* Start accepting connections */
      listen(sock, 5);
      do {
            msgsock = accept(sock, 0, 0);
            if (msgsock == -1)
                  perror("accept");
            else do {
                  bzero(buf, sizeof(buf));
                  if ((rval = read(msgsock, buf, 1024)) < 0)
                        perror("reading stream message");
                  i = 0;
                  if (rval == 0)
                        printf("Ending connection\n");
                  else
                        printf("-->%s\n", buf);
            } while (rval != 0);
            close(msgsock);
      } while (TRUE);
      /*
       * Since this program has an infinite loop, the socket "sock" is
       * never explicitly closed.  However, all sockets will be closed
       * automatically when a process is killed or terminates normally.
       */
}
```

Figure 7b Accepting an Internet domain stream connection

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1

/*
 * This program uses select() to check that someone is trying to connect
 * before calling accept().
 */

main()
{
```

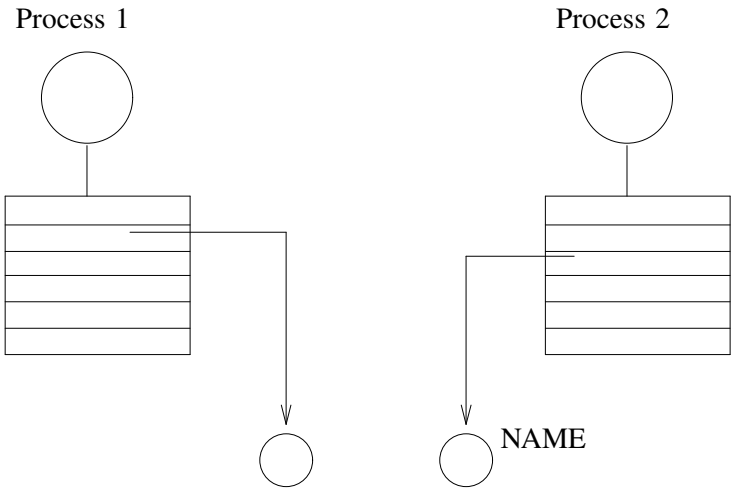
```
int sock, length;
struct sockaddr_in server;
int msgsock;
char buf[1024];
int rval;
fd_set ready;
struct timeval to;

/* Create socket */
sock = socket(AF_INET, SOCK_STREAM, 0);
if (sock < 0) {
      perror("opening stream socket");
      exit(1);
}
/* Name socket using wildcards */
server.sin_family = AF_INET;
server.sin_addr.s_addr = INADDR_ANY;
server.sin_port = 0;
if (bind(sock, &server, sizeof(server))) {
      perror("binding stream socket");
      exit(1);
}
/* Find out assigned port number and print it out */
length = sizeof(server);
if (getsockname(sock, &server, &length)) {
      perror("getting socket name");
      exit(1);
}
printf("Socket has port #%d\n", ntohs(server.sin_port));

/* Start accepting connections */
listen(sock, 5);
do {
      FD_ZERO(&ready);
      FD_SET(sock, &ready);
      to.tv_sec = 5;
      if (select(sock + 1, &ready, 0, 0, &to) < 0) {
            perror("select");
            continue;
      }
      if (FD_ISSET(sock, &ready)) {
            msgsock = accept(sock, (struct sockaddr *)0, (int *)0);
            if (msgsock == -1)
                  perror("accept");
            else do {
                  bzero(buf, sizeof(buf));
                  if ((rval = read(msgsock, buf, 1024)) < 0)
                        perror("reading stream message");
                  else if (rval == 0)
                        printf("Ending connection\n");
                  else
                        printf("-->%s\n", buf);
            } while (rval > 0);
```

```
                close(msgsock);
            } else
                printf("Do something else\n");
        } while (TRUE);
}
```

Figure 7c Using select() to check for pending connections



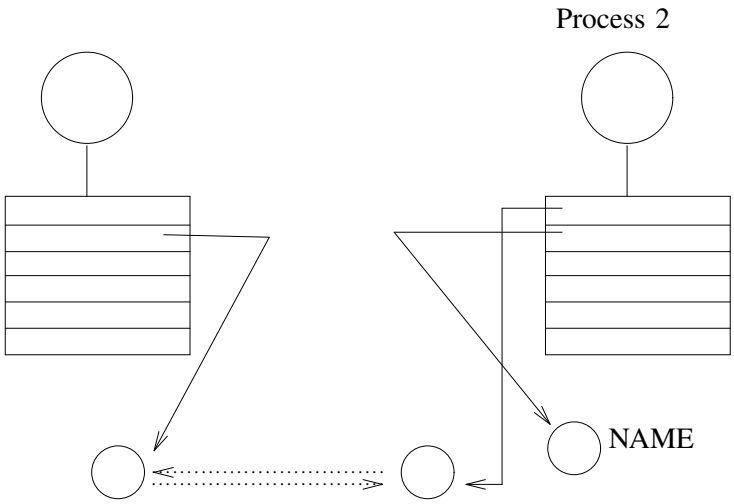


Figure 8 Establishing a stream connection

```
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define DATA "Half a league, half a league . . ."

/*
 * This program connects to the socket named in the command line and sends a
 * one line message to that socket. The form of the command line is
```

```
 * ustreamwrite pathname
 */
main(argc, argv)
      int argc;
      char *argv[];
{
      int sock;
      struct sockaddr_un server;
      char buf[1024];

      /* Create socket */
      sock = socket(AF_UNIX, SOCK_STREAM, 0);
      if (sock < 0) {
            perror("opening stream socket");
            exit(1);
      }
      /* Connect socket using name specified by command line. */
      server.sun_family = AF_UNIX;
      strcpy(server.sun_path, argv[1]);

      if (connect(sock, &server, sizeof(struct sockaddr_un)) < 0) {
            close(sock);
            perror("connecting stream socket");
            exit(1);
      }
      if (write(sock, DATA, sizeof(DATA)) < 0)
            perror("writing on stream socket");
}
```

Figure 9a Initiating a UNIX domain stream connection

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define NAME "socket"

/*
 * This program creates a socket in the UNIX domain and binds a name to it.
 * After printing the socket's name it begins a loop. Each time through the
 * loop it accepts a connection and prints out messages from it.  When the
 * connection breaks, or a termination message comes through, the program
 * accepts a new connection.
 */
main()
{
      int sock, msgsock, rval;
      struct sockaddr_un server;
      char buf[1024];

      /* Create socket */
      sock = socket(AF_UNIX, SOCK_STREAM, 0);
```

```
        if (sock < 0) {
                perror("opening stream socket");
                exit(1);
        }
        /* Name socket using file system name */
        server.sun_family = AF_UNIX;
        strcpy(server.sun_path, NAME);
        if (bind(sock, &server, sizeof(struct sockaddr_un))) {
                perror("binding stream socket");
                exit(1);
        }
        printf("Socket has name %s\n", server.sun_path);
        /* Start accepting connections */
        listen(sock, 5);
        for (;;) {
                msgsock = accept(sock, 0, 0);
                if (msgsock == -1)
                        perror("accept");
                else do {
                        bzero(buf, sizeof(buf));
                        if ((rval = read(msgsock, buf, 1024)) < 0)
                                perror("reading stream message");
                        else if (rval == 0)
                                printf("Ending connection\n");
                        else
                                printf("-->%s\n", buf);
                } while (rval > 0);
                close(msgsock);
        }
        /*
         * The following statements are not executed, because they follow an
         * infinite loop.  However, most ordinary programs will not run
         * forever.  In the UNIX domain it is necessary to tell the file
         * system that one is through using NAME.  In most programs one uses
         * the call unlink() as below. Since the user will have to kill this
         * program, it will be necessary to remove the name by a command from
         * the shell.
         */
        close(sock);
        unlink(NAME);
}
```

Figure 9b Accepting a UNIX domain stream connection

```
         * The variable descriptor may be the descriptor of either a file
         * or of a socket.
         */
        cc = read(descriptor, buf, nbytes)
        int cc, descriptor; char *buf; int nbytes;

        /*
         * An iovec can include several source buffers.
         */
        cc = readv(descriptor, iov, iovcnt)
```

```
int cc, descriptor; struct iovec *iov; int iovcnt;

cc = write(descriptor, buf, nbytes)
int cc, descriptor; char *buf; int nbytes;

cc = writev(descriptor, iovec, ioveclen)
int cc, descriptor; struct iovec *iovec; int ioveclen;

/*
 * The variable ``sock'' must be the descriptor of a socket.
 * Flags may include MSG_OOB and MSG_PEEK.
 */
cc = send(sock, msg, len, flags)
int cc, sock; char *msg; int len, flags;

cc = sendto(sock, msg, len, flags, to, tolen)
int cc, sock; char *msg; int len, flags;
struct sockaddr *to; int tolen;

cc = sendmsg(sock, msg, flags)
int cc, sock; struct msghdr msg[]; int flags;

cc = recv(sock, buf, len, flags)
int cc, sock; char *buf; int len, flags;

cc = recvfrom(sock, buf, len, flags, from, fromlen)
int cc, sock; char *buf; int len, flags;
struct sockaddr *from; int *fromlen;

cc = recvmsg(sock, msg, flags)
int cc, socket; struct msghdr msg[]; int flags;
```

Figure 10 Varieties of read and write commands

An Advanced 4.4BSD Interprocess Communication Tutorial

Samuel J. Leffler

Robert S. Fabry

William N. Joy

Phil Lapsley

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

Steve Miller

Chris Torek

Heterogeneous Systems Laboratory
Department of Computer Science
University of Maryland, College Park
College Park, Maryland 20742

ABSTRACT

This document provides an introduction to the interprocess communication facilities included in the 4.4BSD release of the UNIX\$1 system.

It discusses the overall model for interprocess communication and introduces the interprocess communication primitives which have been added to the system. The majority of the document considers the use of these primitives in developing applications. The reader is expected to be familiar with the C programming language as all examples are written in C.

* UNIX is a trademark of UNIX System Laboratories, Inc. in the US and some other countries.

1. INTRODUCTION

One of the most important additions to UNIX in 4.2BSD was interprocess communication. These facilities were the result of more than two years of discussion and research. The facilities provided in 4.2BSD incorporated many of the ideas from current research, while trying to maintain the UNIX philosophy of simplicity and conciseness. The 4.3BSD release of Berkeley UNIX improved upon some of the IPC facilities while providing an upward-compatible interface. 4.4BSD adds support for ISO protocols and IP multicasting. The BSD interprocess communication facilities have become a defacto standard for UNIX.

UNIX has previously been very weak in the area of interprocess communication. Prior to the 4BSD facilities, the only standard mechanism which allowed two processes to communicate were pipes (the mpx files which were part of Version 7 were experimental). Unfortunately, pipes are very restrictive in that the two communicating processes must be related through a common ancestor. Further, the semantics of pipes makes them almost impossible to maintain in a distributed environment.

Earlier attempts at extending the IPC facilities of UNIX have met with mixed reaction. The majority of the problems have been related to the fact that these facilities have been tied to the UNIX file system, either through naming or implementation. Consequently, the IPC facilities provided in 4.2BSD were designed as a totally independent subsystem. The BSD IPC allows processes to rendezvous in many ways. Processes may rendezvous through a UNIX file system-like name space (a space where all names are path names) as well as through a network name space. In fact, new name spaces may be added at a future time with only minor changes visible to users. Further, the communication facilities have been extended to include more than the simple byte stream provided by a pipe. These extensions have resulted in a completely new part of the system which users will need time to familiarize themselves with. It is likely that as more use is made of these facilities they will be refined; only time will tell.

This document provides a high-level description of the IPC facilities in 4.4BSD and their use. It is designed to complement the manual pages for the IPC primitives by examples of their use. The remainder of this document is organized in four sections. Section 2 introduces the IPC-related system calls and the basic model of communication. Section 3 describes some of the supporting library routines users may find useful in constructing distributed applications. Section 4 is concerned with the client/server model used in developing applications and includes examples of the two major types of servers. Section 5 delves into advanced topics which sophisticated users are likely to encounter when using the IPC facilities.

2. BASICS

The basic building block for communication is the *socket*. A socket is an endpoint of communication to which a name may be *bound*. Each socket in use has a *type* and one or more associated processes. Sockets exist within *communication domains*. A communication domain is an abstraction introduced to bundle common properties of processes communicating through sockets. One such property is the scheme used to name sockets. For example, in the UNIX communication domain sockets are named with UNIX path names; e.g. a socket may be named ''/dev/foo''. Sockets normally exchange data only with sockets in the same domain (it may be possible to cross domain boundaries, but only if some translation process is performed). The 4.4BSD IPC facilities support four separate communication domains: the UNIX domain, for on-system communication; the Internet domain, which is used by processes which communicate using the Internet standard communication protocols; the NS domain, which is used by processes which communicate using the Xerox standard communication protocols*;

* See *Internet Transport Protocols*, Xerox System Integration Standard (XSIS)028112 for more information. This document is almost a necessity for one trying to write NS applications.

and the ISO OSI protocols, which are not documented in this tutorial. The underlying communication facilities provided by these domains have a significant influence on the internal system implementation as well as the interface to socket facilities available to a user. An example of the latter is that a socket "operating" in the UNIX domain sees a subset of the error conditions which are possible when operating in the Internet (or NS) domain.

2.1. Socket types

Sockets are typed according to the communication properties visible to a user. Processes are presumed to communicate only between sockets of the same type, although there is nothing that prevents communication between sockets of different types should the underlying communication protocols support this.

Four types of sockets currently are available to a user. A *stream* socket provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. Aside from the bidirectionality of data flow, a pair of connected stream sockets provides an interface nearly identical to that of pipes†.

A *datagram* socket supports bidirectional flow of data which is not promised to be sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket may find messages duplicated, and, possibly, in an order different from the order in which it was sent. An important characteristic of a datagram socket is that record boundaries in data are preserved. Datagram sockets closely model the facilities found in many contemporary packet switched networks such as the Ethernet.

A *raw* socket provides users access to the underlying communication protocols which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more esoteric facilities of an existing protocol. The use of raw sockets is considered in section 5.

A *sequenced packet* socket is similar to a stream socket, with the exception that record boundaries are preserved. This interface is provided only as part of the NS socket abstraction, and is very important in most serious NS applications. Sequenced-packet sockets allow the user to manipulate the SPP or IDP headers on a packet or a group of packets either by writing a prototype header along with whatever data is to be sent, or by specifying a default header to be used with all outgoing data, and allows the user to receive the headers on incoming packets. The use of these options is considered in section 5.

Another potential socket type which has interesting properties is the *reliably delivered message* socket. The reliably delivered message socket has similar properties to a datagram socket, but with reliable delivery. There is currently no support for this type of socket, but a reliably delivered message protocol similar to Xerox's Packet Exchange Protocol (PEX) may be simulated at the user level. More information on this topic can be found in section 5.

2.2. Socket creation

To create a socket the *socket* system call is used:

```
s = socket(domain, type, protocol);
```

This call requests that the system create a socket in the specified *domain* and of the specified *type*. A particular protocol may also be requested. If the protocol is left unspecified (a value of 0), the system will select an appropriate protocol from those protocols which comprise the communication domain and which may be used to support the requested socket type. The user is returned a descriptor (a small integer number) which may be used in later system calls which operate on sockets. The domain is

† In the UNIX domain, in fact, the semantics are identical and, as one might expect, pipes have been implemented internally as simply a pair of connected stream sockets.

specified as one of the manifest constants defined in the file <*sys/socket.h*>. For the UNIX domain the constant is AF_UNIX*; for the Internet domain AF_INET; and for the NS domain, AF_NS. The socket types are also defined in this file and one of SOCK_STREAM, SOCK_DGRAM, SOCK_RAW, or SOCK_SEQPACKET must be specified. To create a stream socket in the Internet domain the following call might be used:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

This call would result in a stream socket being created with the TCP protocol providing the underlying communication support. To create a datagram socket for on-machine use the call might be:

```
s = socket(AF_UNIX, SOCK_DGRAM, 0);
```

The default protocol (used when the *protocol* argument to the *socket* call is 0) should be correct for most every situation. However, it is possible to specify a protocol other than the default; this will be covered in section 5.

There are several reasons a socket call may fail. Aside from the rare occurrence of lack of memory (ENOBUFS), a socket request may fail due to a request for an unknown protocol (EPROTONOSUPPORT), or a request for a type of socket for which there is no supporting protocol (EPROTOTYPE).

2.3. Binding local names

A socket is created without a name. Until a name is bound to a socket, processes have no way to reference it and, consequently, no messages may be received on it. Communicating processes are bound by an *association*. In the Internet and NS domains, an association is composed of local and foreign addresses, and local and foreign ports, while in the UNIX domain, an association is composed of local and foreign path names (the phrase ''foreign pathname'' means a pathname created by a foreign process, not a pathname on a foreign system). In most domains, associations must be unique. In the Internet domain there may never be duplicate <protocol, local address, local port, foreign address, foreign port> tuples. UNIX domain sockets need not always be bound to a name, but when bound there may never be duplicate <protocol, local pathname, foreign pathname> tuples. The pathnames may not refer to files already existing on the system in 4.3; the situation may change in future releases.

The *bind* system call allows a process to specify half of an association, <local address, local port> (or <local pathname>), while the *connect* and *accept* primitives are used to complete a socket's association.

In the Internet domain, binding names to sockets can be fairly complex. Fortunately, it is usually not necessary to specifically bind an address and port number to a socket, because the *connect* and *send* calls will automatically bind an appropriate address if they are used with an unbound socket. The process of binding names to NS sockets is similar in most ways to that of binding names to Internet sockets.

The *bind* system call is used as follows:

```
bind(s, name, namelen);
```

The bound name is a variable length byte string which is interpreted by the supporting protocol(s). Its interpretation may vary from communication domain to communication domain (this is one of the properties which comprise the ''domain''). As mentioned, in the Internet domain names contain an Internet address and port number. NS domain names contain an NS address and port number. In the UNIX domain, names contain a path name and a family, which is always AF_UNIX. If one wanted to bind the name ''/tmp/foo'' to a UNIX domain socket, the following code would be used*:

---

* The manifest constants are named AF_whatever as they indicate the ''address format'' to use in interpreting names.

* Note that, although the tendency here is to call the ''addr'' structure ''sun'', doing so would cause problems if the code were ever ported to a Sun workstation.

```
#include <sys/un.h>
 ...
struct sockaddr_un addr;
 ...
strcpy(addr.sun_path, "/tmp/foo");
addr.sun_family = AF_UNIX;
bind(s, (struct sockaddr *) &addr, strlen(addr.sun_path) +
    sizeof (addr.sun_len) + sizeof (addr.sun_family));
```

Note that in determining the size of a UNIX domain address null bytes are not counted, which is why *strlen* is used. In the current implementation of UNIX domain IPC, the file name referred to in *addr.sun_path* is created as a socket in the system file space. The caller must, therefore, have write permission in the directory where *addr.sun_path* is to reside, and this file should be deleted by the caller when it is no longer needed. Future versions of 4BSD may not create this file.

In binding an Internet address things become more complicated. The actual call is similar,

```
#include <sys/types.h>
#include <netinet/in.h>
 ...
struct sockaddr_in sin;
 ...
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

but the selection of what to place in the address *sin* requires some discussion. We will come back to the problem of formulating Internet addresses in section 3 when the library routines used in name resolution are discussed.

Binding an NS address to a socket is even more difficult, especially since the Internet library routines do not work with NS hostnames. The actual call is again similar:

```
#include <sys/types.h>
#include <netns/ns.h>
 ...
struct sockaddr_ns sns;
 ...
bind(s, (struct sockaddr *) &sns, sizeof (sns));
```

Again, discussion of what to place in a ‘‘struct sockaddr_ns’’ will be deferred to section 3.

2.4. Connection establishment

Connection establishment is usually asymmetric, with one process a ‘‘client’’ and the other a ‘‘server’’. The server, when willing to offer its advertised services, binds a socket to a well-known address associated with the service and then passively ‘‘listens’’ on its socket. It is then possible for an unrelated process to rendezvous with the server. The client requests services from the server by initiating a ‘‘connection’’ to the server’s socket. On the client side the *connect* call is used to initiate a connection. Using the UNIX domain, this might appear as,

```
struct sockaddr_un server;
 ...
connect(s, (struct sockaddr *)&server, strlen(server.sun_path) +
    sizeof (server.sun_family));
```

while in the Internet domain,

```
struct sockaddr_in server;
 ...
connect(s, (struct sockaddr *)&server, sizeof (server));
```

and in the NS domain,

```
struct sockaddr_ns server;
 ...
connect(s, (struct sockaddr *)&server, sizeof (server));
```

where *server* in the example above would contain either the UNIX pathname, Internet address and port number, or NS address and port number of the server to which the client process wishes to speak. If the client process's socket is unbound at the time of the connect call, the system will automatically select and bind a name to the socket if necessary; c.f. section 5.4. This is the usual way that local addresses are bound to a socket.

An error is returned if the connection was unsuccessful (any name automatically bound by the system, however, remains). Otherwise, the socket is associated with the server and data transfer may begin. Some of the more common errors returned when a connection attempt fails are:

ETIMEDOUT
: After failing to establish a connection for a period of time, the system decided there was no point in retrying the connection attempt any more. This usually occurs because the destination host is down, or because problems in the network resulted in transmissions being lost.

ECONNREFUSED
: The host refused service for some reason. This is usually due to a server process not being present at the requested name.

ENETDOWN or EHOSTDOWN
: These operational errors are returned based on status information delivered to the client host by the underlying communication services.

ENETUNREACH or EHOSTUNREACH
: These operational errors can occur either because the network or host is unknown (no route to the network or host is present), or because of status information returned by intermediate gateways or switching nodes. Many times the status returned is not sufficient to distinguish a network being down from a host being down, in which case the system indicates the entire network is unreachable.

For the server to receive a client's connection it must perform two steps after binding its socket. The first is to indicate a willingness to listen for incoming connection requests:

```
listen(s, 5);
```

The second parameter to the *listen* call specifies the maximum number of outstanding connections which may be queued awaiting acceptance by the server process; this number may be limited by the system. Should a connection be requested while the queue is full, the connection will not be refused, but rather the individual messages which comprise the request will be ignored. This gives a harried server time to make room in its pending connection queue while the client retries the connection request. Had the connection been returned with the ECONNREFUSED error, the client would be unable to tell if the server was up or not. As it is now it is still possible to get the ETIMEDOUT error back, though this is unlikely. The backlog figure supplied with the listen call is currently limited by the system to a maximum of 5 pending connections on any one queue. This avoids the problem of processes hogging system resources by setting an infinite backlog, then ignoring all connection requests.

With a socket marked as listening, a server may *accept* a connection:

```
struct sockaddr_in from;
 ...
fromlen = sizeof (from);
newsock = accept(s, (struct sockaddr *)&from, &fromlen);
```

(For the UNIX domain, *from* would be declared as a *struct sockaddr_un*, and for the NS domain, *from* would be declared as a *struct sockaddr_ns*, but nothing different would need to be done as far as *fromlen* is concerned. In the examples which follow, only Internet routines will be discussed.) A new descriptor is returned on receipt of a connection (along with a new socket). If the server wishes to find out who its client is, it may supply a buffer for the client socket's name. The value-result parameter *fromlen* is initialized by the server to indicate how much space is associated with *from*, then modified on return to reflect the true size of the name. If the client's name is not of interest, the second parameter may be a null pointer.

Accept normally blocks. That is, *accept* will not return until a connection is available or the system call is interrupted by a signal to the process. Further, there is no way for a process to indicate it will accept connections from only a specific individual, or individuals. It is up to the user process to consider who the connection is from and close down the connection if it does not wish to speak to the process. If the server process wants to accept connections on more than one socket, or wants to avoid blocking on the accept call, there are alternatives; they will be considered in section 5.

2.5. Data transfer

With a connection established, data may begin to flow. To send and receive data there are a number of possible calls. With the peer entity at each end of a connection anchored, a user can send or receive a message without specifying the peer. As one might expect, in this case, then the normal *read* and *write* system calls are usable,

```
write(s, buf, sizeof (buf));
read(s, buf, sizeof (buf));
```

In addition to *read* and *write*, the new calls *send* and *recv* may be used:

```
send(s, buf, sizeof (buf), flags);
recv(s, buf, sizeof (buf), flags);
```

While *send* and *recv* are virtually identical to *read* and *write*, the extra *flags* argument is important. The flags, defined in <*sys/socket.h*>, may be specified as a non-zero value if one or more of the following is required:

MSG_OOB	send/receive out of band data
MSG_PEEK	look at data without reading
MSG_DONTROUTE	send data without routing packets

Out of band data is a notion specific to stream sockets, and one which we will not immediately consider. The option to have data sent without routing applied to the outgoing packets is currently used only by the routing table management process, and is unlikely to be of interest to the casual user. The ability to preview data is, however, of interest. When MSG_PEEK is specified with a *recv* call, any data present is returned to the user, but treated as still ''unread''. That is, the next *read* or *recv* call applied to the socket will return the data previously previewed.

2.6. Discarding sockets

Once a socket is no longer of interest, it may be discarded by applying a *close* to the descriptor,

```
close(s);
```

If data is associated with a socket which promises reliable delivery (e.g. a stream socket) when a close takes place, the system will continue to attempt to transfer the data. However, after a fairly long period of time, if the data is still undelivered, it will be discarded. Should a user have no use for any pending data, it may perform a *shutdown* on the socket prior to closing it. This call is of the form:

```
shutdown(s, how);
```

where *how* is 0 if the user is no longer interested in reading data, 1 if no more data will be sent, or 2 if no data is to be sent or received.

2.7. Connectionless sockets

To this point we have been concerned mostly with sockets which follow a connection oriented model. However, there is also support for connectionless interactions typical of the datagram facilities found in contemporary packet switched networks. A datagram socket provides a symmetric interface to data exchange. While processes are still likely to be client and server, there is no requirement for connection establishment. Instead, each message includes the destination address.

Datagram sockets are created as before. If a particular local address is needed, the *bind* operation must precede the first data transmission. Otherwise, the system will set the local address and/or port when data is first sent. To send data, the *sendto* primitive is used,

```
sendto(s, buf, buflen, flags, (struct sockaddr *)&to, tolen);
```

The *s*, *buf*, *buflen*, and *flags* parameters are used as before. The *to* and *tolen* values are used to indicate the address of the intended recipient of the message. When using an unreliable datagram interface, it is unlikely that any errors will be reported to the sender. When information is present locally to recognize a message that can not be delivered (for instance when a network is unreachable), the call will return –1 and the global value *errno* will contain an error number.

To receive messages on an unconnected datagram socket, the *recvfrom* primitive is provided:

```
recvfrom(s, buf, buflen, flags, (struct sockaddr *)&from, &fromlen);
```

Once again, the *fromlen* parameter is handled in a value-result fashion, initially containing the size of the *from* buffer, and modified on return to indicate the actual size of the address from which the datagram was received.

In addition to the two calls mentioned above, datagram sockets may also use the *connect* call to associate a socket with a specific destination address. In this case, any data sent on the socket will automatically be addressed to the connected peer, and only data received from that peer will be delivered to the user. Only one connected address is permitted for each socket at one time; a second connect will change the destination address, and a connect to a null address (family AF_UNSPEC) will disconnect. Connect requests on datagram sockets return immediately, as this simply results in the system recording the peer's address (as compared to a stream socket, where a connect request initiates establishment of an end to end connection). *Accept* and *listen* are not used with datagram sockets.

While a datagram socket socket is connected, errors from recent *send* calls may be returned asynchronously. These errors may be reported on subsequent operations on the socket, or a special socket option used with *getsockopt*, SO_ERROR, may be used to interrogate the error status. A *select* for reading or writing will return true when an error indication has been received. The next operation will return the error, and the error status is cleared. Other of the less important details of datagram sockets are described in section 5.

2.8. Input/Output multiplexing

One last facility often used in developing applications is the ability to multiplex i/o requests among multiple sockets and/or files. This is done using the *select* call:

```
#include <sys/time.h>
#include <sys/types.h>
 ...

fd_set readmask, writemask, exceptmask;
struct timeval timeout;
 ...
select(nfds, &readmask, &writemask, &exceptmask, &timeout);
```

Select takes as arguments pointers to three sets, one for the set of file descriptors for which the caller wishes to be able to read data on, one for those descriptors to which data is to be written, and one for which exceptional conditions are pending; out-of-band data is the only exceptional condition currently implemented by the socket If the user is not interested in certain conditions (i.e., read, write, or exceptions), the corresponding argument to the *select* should be a null pointer.

Each set is actually a structure containing an array of long integer bit masks; the size of the array is set by the definition FD_SETSIZE. The array is be long enough to hold one bit for each of FD_SETSIZE file descriptors.

The macros FD_SET(*fd, &mask*) and FD_CLR(*fd, &mask*) have been provided for adding and removing file descriptor *fd* in the set *mask*. The set should be zeroed before use, and the macro FD_ZERO(*&mask*) has been provided to clear the set *mask*. The parameter *nfds* in the *select* call specifies the range of file descriptors (i.e. one plus the value of the largest descriptor) to be examined in a set.

A timeout value may be specified if the selection is not to last more than a predetermined period of time. If the fields in *timeout* are set to 0, the selection takes the form of a *poll*, returning immediately. If the last parameter is a null pointer, the selection will block indefinitely*. *Select* normally returns the number of file descriptors selected; if the *select* call returns due to the timeout expiring, then the value 0 is returned. If the *select* terminates because of an error or interruption, a –1 is returned with the error number in *errno*, and with the file descriptor masks unchanged.

Assuming a successful return, the three sets will indicate which file descriptors are ready to be read from, written to, or have exceptional conditions pending. The status of a file descriptor in a select mask may be tested with the *FD_ISSET(fd, &mask)* macro, which returns a non-zero value if *fd* is a member of the set *mask*, and 0 if it is not.

To determine if there are connections waiting on a socket to be used with an *accept* call, *select* can be used, followed by a *FD_ISSET(fd, &mask)* macro to check for read readiness on the appropriate socket. If *FD_ISSET* returns a non-zero value, indicating permission to read, then a connection is pending on the socket.

As an example, to read data from two sockets, *s1* and *s2* as it is available from each and with a one-second timeout, the following code might be used:

---

* To be more specific, a return takes place only when a descriptor is selectable, or when a signal is received by the caller, interrupting the system call.

```
#include <sys/time.h>
#include <sys/types.h>
 ...
fd_set read_template;
struct timeval wait;
 ...
for (;;) {
        wait.tv_sec = 1;            /* one second */
        wait.tv_usec = 0;

        FD_ZERO(&read_template);

        FD_SET(s1, &read_template);
        FD_SET(s2, &read_template);

        nb = select(FD_SETSIZE, &read_template, (fd_set *) 0, (fd_set *) 0, &wait);
        if (nb <= 0) {
                An error occurred during the select, or
                the select timed out.
        }

        if (FD_ISSET(s1, &read_template)) {
                Socket #1 is ready to be read from.
        }

        if (FD_ISSET(s2, &read_template)) {
                Socket #2 is ready to be read from.
        }
}
```

In 4.2, the arguments to *select* were pointers to integers instead of pointers to *fd_set*s. This type of call will still work as long as the number of file descriptors being examined is less than the number of bits in an integer; however, the methods illustrated above should be used in all current programs.

Select provides a synchronous multiplexing scheme. Asynchronous notification of output completion, input availability, and exceptional conditions is possible through use of the SIGIO and SIGURG signals described in section 5.

3. NETWORK LIBRARY ROUTINES

The discussion in section 2 indicated the possible need to locate and construct network addresses when using the interprocess communication facilities in a distributed environment. To aid in this task a number of routines have been added to the standard C run-time library. In this section we will consider the new routines provided to manipulate network addresses. While the 4.4BSD networking facilities support the Internet protocols and the Xerox NS protocols, most of the routines presented in this section do not apply to the NS domain. Unless otherwise stated, it should be assumed that the routines presented in this section do not apply to the NS domain.

Locating a service on a remote host requires many levels of mapping before client and server may communicate. A service is assigned a name which is intended for human consumption; e.g. ‘‘the *login server* on host monet’’. This name, and the name of the peer host, must then be translated into network

addresses which are not necessarily suitable for human consumption. Finally, the address must then used in locating a physical *location* and *route* to the service. The specifics of these three mappings are likely to vary between network architectures. For instance, it is desirable for a network to not require hosts to be named in such a way that their physical location is known by the client host. Instead, underlying services in the network may discover the actual location of the host at the time a client host wishes to communicate. This ability to have hosts named in a location independent manner may induce overhead in connection establishment, as a discovery process must take place, but allows a host to be physically mobile without requiring it to notify its clientele of its current location.

Standard routines are provided for: mapping host names to network addresses, network names to network numbers, protocol names to protocol numbers, and service names to port numbers and the appropriate protocol to use in communicating with the server process. The file <*netdb.h*> must be included when using any of these routines.

3.1. Host names

An Internet host name to address mapping is represented by the *hostent* structure:

```
struct    hostent {
          char    *h_name;          /* official name of host */
          char    **h_aliases;      /* alias list */
          int     h_addrtype;       /* host address type (e.g., AF_INET) */
          int     h_length;         /* length of address */
          char    **h_addr_list;    /* list of addresses, null terminated */
};

#define   h_addr  h_addr_list[0]    /* first address, network byte order */
```

The routine *gethostbyname*(3N) takes an Internet host name and returns a *hostent* structure, while the routine *gethostbyaddr*(3N) maps Internet host addresses into a *hostent* structure.

The official name of the host and its public aliases are returned by these routines, along with the address type (family) and a null terminated list of variable length address. This list of addresses is required because it is possible for a host to have many addresses, all having the same name. The *h_addr* definition is provided for backward compatibility, and is defined to be the first address in the list of addresses in the *hostent* structure.

The database for these calls is provided either by the file */etc/hosts* (*hosts*(5)), or by use of a nameserver, *named*(8). Because of the differences in these databases and their access protocols, the information returned may differ. When using the host table version of *gethostbyname*, only one address will be returned, but all listed aliases will be included. The nameserver version may return alternate addresses, but will not provide any aliases other than one given as argument.

Unlike Internet names, NS names are always mapped into host addresses by the use of a standard NS *Clearinghouse service*, a distributed name and authentication server. The algorithms for mapping NS names to addresses via a Clearinghouse are rather complicated, and the routines are not part of the standard libraries. The user-contributed Courier (Xerox remote procedure call protocol) compiler contains routines to accomplish this mapping; see the documentation and examples provided therein for more information. It is expected that almost all software that has to communicate using NS will need to use the facilities of the Courier compiler.

An NS host address is represented by the following:

```
union ns_host {
        u_char      c_host[6];
        u_short     s_host[3];
};

union ns_net {
        u_char      c_net[4];
        u_short     s_net[2];
};

struct ns_addr {
        union ns_net        x_net;
        union ns_host       x_host;
        u_short     x_port;
};
```

The following code fragment inserts a known NS address into a *ns_addr*:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
 ...
u_long netnum;
struct sockaddr_ns dst;
 ...
bzero((char *)&dst, sizeof(dst));

/*
 * There is no convenient way to assign a long
 * integer to a ‘‘union ns_net’’ at present; in
 * the future, something will hopefully be provided,
 * but this is the portable way to go for now.
 * The network number below is the one for the NS net
 * that the desired host (gyre) is on.
 */
netnum = htonl(2266);
dst.sns_addr.x_net = *(union ns_net *) &netnum;
dst.sns_family = AF_NS;

/*
 * host 2.7.1.0.2a.18 == "gyre:Computer Science:UofMaryland"
 */
dst.sns_addr.x_host.c_host[0] = 0x02;
dst.sns_addr.x_host.c_host[1] = 0x07;
dst.sns_addr.x_host.c_host[2] = 0x01;
dst.sns_addr.x_host.c_host[3] = 0x00;
dst.sns_addr.x_host.c_host[4] = 0x2a;
dst.sns_addr.x_host.c_host[5] = 0x18;
dst.sns_addr.x_port = htons(75);
```

3.2. Network names

As for host names, routines for mapping network names to numbers, and back, are provided. These routines return a *netent* structure:

```
/*
 * Assumption here is that a network number
 * fits in 32 bits -- probably a poor one.
 */
struct  netent {
        char        *n_name;        /* official name of net */
        char        **n_aliases;    /* alias list */
        int         n_addrtype;     /* net address type */
        int         n_net;          /* network number, host byte order */
};
```

The routines *getnetbyname*(3N), *getnetbynumber*(3N), and *getnetent*(3N) are the network counterparts to the host routines described above. The routines extract their information from */etc/networks*.

NS network numbers are determined either by asking your local Xerox Network Administrator (and hardcoding the information into your code), or by querying the Clearinghouse for addresses. The internetwork router is the only process that needs to manipulate network numbers on a regular basis; if a process wishes to communicate with a machine, it should ask the Clearinghouse for that machine's address (which will include the net number).

3.3. Protocol names

For protocols, which are defined in */etc/protocols*, the *protoent* structure defines the protocol-name mapping used with the routines *getprotobyname*(3N), *getprotobynumber*(3N), and *getprotoent*(3N):

```
struct  protoent {
        char        *p_name;        /* official protocol name */
        char        **p_aliases;    /* alias list */
        int         p_proto;        /* protocol number */
};
```

In the NS domain, protocols are indicated by the "client type" field of a IDP header. No protocol database exists; see section 5 for more information.

3.4. Service names

Information regarding services is a bit more complicated. A service is expected to reside at a specific ‘‘port’’ and employ a particular communication protocol. This view is consistent with the Internet domain, but inconsistent with other network architectures. Further, a service may reside on multiple ports. If this occurs, the higher level library routines will have to be bypassed or extended. Services available are contained in the file */etc/services*. A service mapping is described by the *servent* structure,

```
struct  servent {
        char        *s_name;        /* official service name */
        char        **s_aliases;    /* alias list */
        int         s_port;         /* port number, network byte order */
        char        *s_proto;       /* protocol to use */
};
```

The routine *getservbyname*(3N) maps service names to a servent structure by specifying a service name and, optionally, a qualifying protocol. Thus the call

```
sp = getservbyname("telnet", (char *) 0);
```

returns the service specification for a telnet server using any protocol, while the call

```
sp = getservbyname("telnet", "tcp");
```

returns only that telnet server which uses the TCP protocol. The routines *getservbyport*(3N) and *getservent*(3N) are also provided. The *getservbyport* routine has an interface similar to that provided by *getservbyname*; an optional protocol name may be specified to qualify lookups.

In the NS domain, services are handled by a central dispatcher provided as part of the Courier remote procedure call facilities. Again, the reader is referred to the Courier compiler documentation and to the Xerox standard* for further details.

3.5. Miscellaneous

With the support routines described above, an Internet application program should rarely have to deal directly with addresses. This allows services to be developed as much as possible in a network independent fashion. It is clear, however, that purging all network dependencies is very difficult. So long as the user is required to supply network addresses when naming services and sockets there will always some network dependency in a program. For example, the normal code included in client programs, such as the remote login program, is of the form shown in Figure 1. (This example will be considered in more detail in section 4.)

If we wanted to make the remote login program independent of the Internet protocols and addressing scheme we would be forced to add a layer of routines which masked the network dependent aspects from the mainstream login code. For the current facilities available in the system this does not appear to be worthwhile.

Aside from the address-related data base routines, there are several other routines available in the run-time library which are of interest to users. These are intended mostly to simplify manipulation of names and addresses. Table 1 summarizes the routines for manipulating variable length byte strings and handling byte swapping of network addresses and values.

Call	Synopsis
bcmp(s1, s2, n)	compare byte-strings; 0 if same, not 0 otherwise
bcopy(s1, s2, n)	copy n bytes from s1 to s2
bzero(base, n)	zero-fill n bytes starting at base
htonl(val)	convert 32-bit quantity from host to network byte order
htons(val)	convert 16-bit quantity from host to network byte order
ntohl(val)	convert 32-bit quantity from network to host byte order
ntohs(val)	convert 16-bit quantity from network to host byte order

Table 1. C run-time routines.

The byte swapping routines are provided because the operating system expects addresses to be supplied in network order (aka ‘‘big-endian’’ order). On ‘‘little-endian’’ architectures, such as Intel x86 and VAX, host byte ordering is different than network byte ordering. Consequently, programs are sometimes required to byte swap quantities. The library routines which return network addresses provide them in network order so that they may simply be copied into the structures provided to the system. This implies users should encounter the byte swapping problem only when *interpreting* network addresses. For example, if an Internet port is to be printed out the following code would be required:

```
printf("port number %d\n", ntohs(sp->s_port));
```

* *Courier: The Remote Procedure Call Protocol*, XSIS 038112.

On machines where unneeded these routines are defined as null macros.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
 ...
main(argc, argv)
        int argc;
        char *argv[];
{
        struct sockaddr_in server;
        struct servent *sp;
        struct hostent *hp;
        int s;
        ...
        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogin: tcp/login: unknown service\n");
                exit(1);
        }
        hp = gethostbyname(argv[1]);
        if (hp == NULL) {
                fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
                exit(2);
        }
        bzero((char *)&server, sizeof (server));
        bcopy(hp->h_addr, (char *)&server.sin_addr, hp->h_length);
        server.sin_family = hp->h_addrtype;
        server.sin_port = sp->s_port;
        s = socket(AF_INET, SOCK_STREAM, 0);
        if (s < 0) {
                perror("rlogin: socket");
                exit(3);
        }
        ...
        /* Connect does the bind() for us */

        if (connect(s, (char *)&server, sizeof (server)) < 0) {
                perror("rlogin: connect");
                exit(5);
        }
        ...
}
```

Figure 1. Remote login client code.

4. CLIENT/SERVER MODEL

The most commonly used paradigm in constructing distributed applications is the client/server model. In this scheme client applications request services from a server process. This implies an asymmetry in establishing communication between the client and server which has been examined in section 2. In this section we will look more closely at the interactions between client and server, and consider some of the problems in developing client and server applications.

The client and server require a well known set of conventions before service may be rendered (and accepted). This set of conventions comprises a protocol which must be implemented at both ends of a connection. Depending on the situation, the protocol may be symmetric or asymmetric. In a symmetric protocol, either side may play the master or slave roles. In an asymmetric protocol, one side is immutably recognized as the master, with the other as the slave. An example of a symmetric protocol is the TELNET protocol used in the Internet for remote terminal emulation. An example of an asymmetric protocol is the Internet file transfer protocol, FTP. No matter whether the specific protocol used in obtaining a service is symmetric or asymmetric, when accessing a service there is a ''client process'' and a ''server process''. We will first consider the properties of server processes, then client processes.

A server process normally listens at a well known address for service requests. That is, the server process remains dormant until a connection is requested by a client's connection to the server's address. At such a time the server process ''wakes up'' and services the client, performing whatever appropriate actions the client requests of it.

Alternative schemes which use a service server may be used to eliminate a flock of server processes clogging the system while remaining dormant most of the time. For Internet servers in 4.4BSD, this scheme has been implemented via *inetd*, the so called ''internet super-server.'' *Inetd* listens at a variety of ports, determined at start-up by reading a configuration file. When a connection is requested to a port on which *inetd* is listening, *inetd* executes the appropriate server program to handle the client. With this method, clients are unaware that an intermediary such as *inetd* has played any part in the connection. *Inetd* will be described in more detail in section 5.

A similar alternative scheme is used by most Xerox services. In general, the Courier dispatch process (if used) accepts connections from processes requesting services of some sort or another. The client processes request a particular <program number, version number, procedure number> triple. If the dispatcher knows of such a program, it is started to handle the request; if not, an error is reported to the client. In this way, only one port is required to service a large variety of different requests. Again, the Courier facilities are not available without the use and installation of the Courier compiler. The information presented in this section applies only to NS clients and services that do not use Courier.

4.1. Servers

In 4.4BSD most servers are accessed at well known Internet addresses or UNIX domain names. For example, the remote login server's main loop is of the form shown in Figure 2.

The first step taken by the server is look up its service definition:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogind: tcp/login: unknown service\n");
        exit(1);
}
```

The result of the *getservbyname* call is used in later portions of the code to define the Internet port at

which it listens for service requests (indicated by a connection).

```
main(argc, argv)
        int argc;
        char *argv[];
{
        int f;
        struct sockaddr_in from;
        struct servent *sp;

        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogind: tcp/login: unknown service\n");
                exit(1);
        }
        ...
#ifndef DEBUG
        /* Disassociate server from controlling terminal */
        ...
#endif

        sin.sin_port = sp->s_port;      /* Restricted port -- see section 5 */
        ...
        f = socket(AF_INET, SOCK_STREAM, 0);
        ...
        if (bind(f, (struct sockaddr *) &sin, sizeof (sin)) < 0) {
                ...
        }
        ...
        listen(f, 5);
        for (;;) {
                int g, len = sizeof (from);

                g = accept(f, (struct sockaddr *) &from, &len);
                if (g < 0) {
                        if (errno != EINTR)
                                syslog(LOG_ERR, "rlogind: accept: %m");
                        continue;
                }
                if (fork() == 0) {
                        close(f);
                        doit(g, &from);
                }
                close(g);
        }
}
```

Figure 2. Remote login server.

Step two is to disassociate the server from the controlling terminal of its invoker:

```
for (i = 0; i < 3; ++i)
        close(i);

open("/", O_RDONLY);
dup2(0, 1);
dup2(0, 2);

i = open("/dev/tty", O_RDWR);
if (i >= 0) {
        ioctl(i, TIOCNOTTY, 0);
        close(i);
}
```

This step is important as the server will likely not want to receive signals delivered to the process group of the controlling terminal. Note, however, that once a server has disassociated itself it can no longer send reports of errors to a terminal, and must log errors via *syslog*.

Once a server has established a pristine environment, it creates a socket and begins accepting service requests. The *bind* call is required to insure the server listens at its expected location. It should be noted that the remote login server listens at a restricted port number, and must therefore be run with a user-id of root. This concept of a ‘‘restricted port number’’ is 4BSD specific, and is covered in section 5.

The main body of the loop is fairly simple:

```
for (;;) {
        int g, len = sizeof (from);

        g = accept(f, (struct sockaddr *)&from, &len);
        if (g < 0) {
                if (errno != EINTR)
                        syslog(LOG_ERR, "rlogind: accept: %m");
                continue;
        }
        if (fork() == 0) { /* Child */
                close(f);
                doit(g, &from);
        }
        close(g);           /* Parent */
}
```

An *accept* call blocks the server until a client requests service. This call could return a failure status if the call is interrupted by a signal such as SIGCHLD (to be discussed in section 5). Therefore, the return value from *accept* is checked to insure a connection has actually been established, and an error report is logged via *syslog* if an error has occurred.

With a connection in hand, the server then forks a child process and invokes the main body of the remote login protocol processing. Note how the socket used by the parent for queuing connection requests is closed in the child, while the socket created as a result of the *accept* is closed in the parent. The address of the client is also handed the *doit* routine because it requires it in authenticating clients.

4.2. Clients

The client side of the remote login service was shown earlier in Figure 1. One can see the separate, asymmetric roles of the client and server clearly in the code. The server is a passive entity,

listening for client connections, while the client process is an active entity, initiating a connection when invoked.

Let us consider more closely the steps taken by the client remote login process. As in the server process, the first step is to locate the service definition for a remote login:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogin: tcp/login: unknown service\n");
        exit(1);
}
```

Next the destination host is looked up with a *gethostbyname* call:

```
hp = gethostbyname(argv[1]);
if (hp == NULL) {
        fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
        exit(2);
}
```

With this accomplished, all that is required is to establish a connection to the server at the requested host and start up the remote login protocol. The address buffer is cleared, then filled in with the Internet address of the foreign host and the port number at which the login process resides on the foreign host:

```
bzero((char *)&server, sizeof (server));
bcopy(hp->h_addr, (char *) &server.sin_addr, hp->h_length);
server.sin_family = hp->h_addrtype;
server.sin_port = sp->s_port;
```

A socket is created, and a connection initiated. Note that *connect* implicitly performs a *bind* call, since *s* is unbound.

```
s = socket(hp->h_addrtype, SOCK_STREAM, 0);
if (s < 0) {
        perror("rlogin: socket");
        exit(3);
}
 ...
if (connect(s, (struct sockaddr *) &server, sizeof (server)) < 0) {
        perror("rlogin: connect");
        exit(4);
}
```

The details of the remote login protocol will not be considered here.

4.3. Connectionless servers

While connection-based services are the norm, some services are based on the use of datagram sockets. One, in particular, is the ‘‘rwho’’ service which provides users with status information for hosts connected to a local area network. This service, while predicated on the ability to *broadcast* information to all hosts connected to a particular network, is of interest as an example usage of datagram sockets.

A user on any machine running the rwho server may find out the current status of a machine with the *ruptime*(1) program. The output generated is illustrated in Figure 3.

Status information for each host is periodically broadcast by rwho server processes on each machine. The same server process also receives the status information and uses it to update a database. This database is then interpreted to generate the status information for each host. Servers operate

```
arpa      up   9:45,      5 users, load  1.15,  1.39,  1.31
cad       up   2+12:04,   8 users, load  4.67,  5.13,  4.59
calder    up   10:10,     0 users, load  0.27,  0.15,  0.14
dali      up   2+06:28,   9 users, load  1.04,  1.20,  1.65
degas     up   25+09:48,  0 users, load  1.49,  1.43,  1.41
ear       up   5+00:05,   0 users, load  1.51,  1.54,  1.56
ernie   down   0:24
esvax   down   17:04
ingres  down   0:26
kim       up   3+09:16,   8 users, load  2.03,  2.46,  3.11
matisse   up   3+06:18,   0 users, load  0.03,  0.03,  0.05
medea     up   3+09:39,   2 users, load  0.35,  0.37,  0.50
merlin  down   19+15:37
miro      up   1+07:20,   7 users, load  4.59,  3.28,  2.12
monet     up   1+00:43,   2 users, load  0.22,  0.09,  0.07
oz      down   16:09
statvax   up   2+15:57,   3 users, load  1.52,  1.81,  1.86
ucbvax    up   9:34,      2 users, load  6.08,  5.16,  3.28
```

Figure 3. ruptime output.

autonomously, coupled only by the local network and its broadcast capabilities.

Note that the use of broadcast for such a task is fairly inefficient, as all hosts must process each message, whether or not using an rwho server. Unless such a service is sufficiently universal and is frequently used, the expense of periodic broadcasts outweighs the simplicity.

Multicasting is an alternative to broadcasting. Setting up multicast sockets is described in Section 5.10.

The rwho server, in a simplified form, is pictured in Figure 4. There are two separate tasks performed by the server. The first task is to act as a receiver of status information broadcast by other hosts on the network. This job is carried out in the main loop of the program. Packets received at the rwho port are interrogated to insure they've been sent by another rwho server process, then are time stamped with their arrival time and used to update a file indicating the status of the host. When a host has not been heard from for an extended period of time, the database interpretation routines assume the host is down and indicate such on the status reports. This algorithm is prone to error as a server may be down while a host is actually up, but serves our current needs.

The second task performed by the server is to supply information regarding the status of its host. This involves periodically acquiring system status information, packaging it up in a message and broadcasting it on the local network for other rwho servers to hear. The supply function is triggered by a timer and runs off a signal. Locating the system status information is somewhat involved, but uninteresting. Deciding where to transmit the resultant packet is somewhat problematical, however.

Status information must be broadcast on the local network. For networks which do not support the notion of broadcast another scheme must be used to simulate or replace broadcasting. One possibility is to enumerate the known neighbors (based on the status messages received from other rwho servers). This, unfortunately, requires some bootstrapping information, for a server will have no idea what machines are its neighbors until it receives status messages from them. Therefore, if all machines on a net are freshly booted, no machine will have any known neighbors and thus never receive, or send, any status information. This is the identical problem faced by the routing table management process in propagating routing status information. The standard solution, unsatisfactory as it may be, is to inform one or more servers of known neighbors and request that they always communicate with these

neighbors. If each server has at least one neighbor supplied to it, status information may then propagate through a neighbor to hosts which are not (possibly) directly neighbors. If the server is able to support networks which provide a broadcast capability, as well as those which do not, then networks with an arbitrary topology may share status information*.

* One must, however, be concerned about ‘‘loops’’. That is, if a host is connected to multiple networks, it will receive status information from itself. This can lead to an endless, wasteful, exchange of information.

```
main()
{
        ...
        sp = getservbyname("who", "udp");
        net = getnetbyname("localnet");
        sin.sin_addr = inet_makeaddr(INADDR_ANY, net);
        sin.sin_port = sp->s_port;
        ...
        s = socket(AF_INET, SOCK_DGRAM, 0);
        ...
        on = 1;
        if (setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on)) < 0) {
                syslog(LOG_ERR, "setsockopt SO_BROADCAST: %m");
                exit(1);
        }
        bind(s, (struct sockaddr *) &sin, sizeof (sin));
        ...
        signal(SIGALRM, onalrm);
        onalrm();
        for (;;) {
                struct whod wd;
                int cc, whod, len = sizeof (from);

                cc = recvfrom(s, (char *)&wd, sizeof (struct whod), 0,
                    (struct sockaddr *)&from, &len);
                if (cc <= 0) {
                        if (cc < 0 && errno != EINTR)
                                syslog(LOG_ERR, "rwhod: recv: %m");
                        continue;
                }
                if (from.sin_port != sp->s_port) {
                        syslog(LOG_ERR, "rwhod: %d: bad from port",
                                ntohs(from.sin_port));
                        continue;
                }
                ...
                if (!verify(wd.wd_hostname)) {
                        syslog(LOG_ERR, "rwhod: malformed host name from %x",
                                ntohl(from.sin_addr.s_addr));
                        continue;
                }
                (void) sprintf(path, "%s/whod.%s", RWHODIR, wd.wd_hostname);
                whod = open(path, O_WRONLY | O_CREAT | O_TRUNC, 0666);
                ...
                (void) time(&wd.wd_recvtime);
                (void) write(whod, (char *)&wd, cc);
                (void) close(whod);
        }
}
```

Figure 4. rwho server.

It is important that software operating in a distributed environment not have any site-dependent information compiled into it. This would require a separate copy of the server at each host and make maintenance a severe headache. 4.4BSD attempts to isolate host-specific information from applications by providing system calls which return the necessary information*. A mechanism exists, in the form of an *ioctl* call, for finding the collection of networks to which a host is directly connected. Further, a local network broadcasting mechanism has been implemented at the socket level. Combining these two features allows a process to broadcast on any directly connected local network which supports the notion of broadcasting in a site independent manner. This allows 4.4BSD to solve the problem of deciding how to propagate status information in the case of *rwho*, or more generally in broadcasting: Such status information is broadcast to connected networks at the socket level, where the connected networks have been obtained via the appropriate *ioctl* calls. The specifics of such broadcastings are complex, however, and will be covered in section 5.

5. ADVANCED TOPICS

A number of facilities have yet to be discussed. For most users of the IPC the mechanisms already described will suffice in constructing distributed applications. However, others will find the need to utilize some of the features which we consider in this section.

5.1. Out of band data

The stream socket abstraction includes the notion of ‘‘out of band’’ data. Out of band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out of band data is delivered to the user independently of normal data. The abstraction defines that the out of band data facilities must support the reliable delivery of at least one out of band message at a time. This message may contain at least one byte of data, and at least one message may be pending delivery to the user at any one time. For communications protocols which support only in-band signaling (i.e. the urgent data is delivered in sequence with the normal data), the system normally extracts the data from the normal data stream and stores it separately. This allows users to choose between receiving the urgent data in order and receiving it out of sequence without having to buffer all the intervening data. It is possible to ‘‘peek’’ (via MSG_PEEK) at out of band data. If the socket has a process group, a SIGURG signal is generated when the protocol is notified of its existence. A process can set the process group or process id to be informed by the SIGURG signal via the appropriate *fcntl* call, as described below for SIGIO. If multiple sockets may have out of band data awaiting delivery, a *select* call for exceptional conditions may be used to determine those sockets with such data pending. Neither the signal nor the select indicate the actual arrival of the out-of-band data, but only notification that it is pending.

In addition to the information passed, a logical mark is placed in the data stream to indicate the point at which the out of band data was sent. The remote login and remote shell applications use this facility to propagate signals between client and server processes. When a signal flushs any pending output from the remote process(es), all data up to the mark in the data stream is discarded.

To send an out of band message the MSG_OOB flag is supplied to a *send* or *sendto* calls, while to receive out of band data MSG_OOB should be indicated when performing a *recvfrom* or *recv* call. To find out if the read pointer is currently pointing at the mark in the data stream, the SIOCATMARK ioctl is provided:

```
ioctl(s, SIOCATMARK, &yes);
```

If *yes* is a 1 on return, the next read will return data after the mark. Otherwise (assuming out of band data has arrived), the next read will provide data sent by the client prior to transmission of the out of

* An example of such a system call is the *gethostname*(2) call which returns the host’s ‘‘official’’ name.

band signal. The routine used in the remote login process to flush output on receipt of an interrupt or quit signal is shown in Figure 5. It reads the normal data up to the mark (to discard it), then reads the out-of-band byte.

```
#include <sys/ioctl.h>
#include <sys/file.h>
 ...
oob()
{
        int out = FWRITE, mark;
        char waste[BUFSIZ];

        /* flush local terminal output */
        ioctl(1, TIOCFLUSH, (char *)&out);
        for (;;) {
                if (ioctl(rem, SIOCATMARK, &mark) < 0) {
                        perror("ioctl");
                        break;
                }
                if (mark)
                        break;
                (void) read(rem, waste, sizeof (waste));
        }
        if (recv(rem, &mark, 1, MSG_OOB) < 0) {
                perror("recv");
                ...
        }
        ...
}
```

Figure 5. Flushing terminal I/O on receipt of out of band data.

A process may also read or peek at the out-of-band data without first reading up to the mark. This is more difficult when the underlying protocol delivers the urgent data in-band with the normal data, and only sends notification of its presence ahead of time (e.g., the TCP protocol used to implement streams in the Internet domain). With such protocols, the out-of-band byte may not yet have arrived when a *recv* is done with the MSG_OOB flag. In that case, the call will return an error of EWOULDBLOCK. Worse, there may be enough in-band data in the input buffer that normal flow control prevents the peer from sending the urgent data until the buffer is cleared. The process must then read enough of the queued data that the urgent data may be delivered.

Certain programs that use multiple bytes of urgent data and must handle multiple urgent signals (e.g., *telnet*(1C)) need to retain the position of urgent data within the stream. This treatment is available as a socket-level option, SO_OOBINLINE; see *setsockopt*(2) for usage. With this option, the position of urgent data (the ''mark'') is retained, but the urgent data immediately follows the mark within the normal data stream returned without the MSG_OOB flag. Reception of multiple urgent indications causes the mark to move, but no out-of-band data are lost.

5.2. Non-Blocking Sockets

It is occasionally convenient to make use of sockets which do not block; that is, I/O requests which cannot complete immediately and would therefore cause the process to be suspended awaiting completion are not executed, and an error code is returned. Once a socket has been created via the *socket* call, it may be marked as non-blocking by *fcntl* as follows:

```
#include <fcntl.h>
 ...
int     s;
 ...
s = socket(AF_INET, SOCK_STREAM, 0);
 ...
if (fcntl(s, F_SETFL, FNDELAY) < 0)
        perror("fcntl F_SETFL, FNDELAY");
        exit(1);
}
 ...
```

When performing non-blocking I/O on sockets, one must be careful to check for the error EWOULDBLOCK (stored in the global variable *errno*), which occurs when an operation would normally block, but the socket it was performed on is marked as non-blocking. In particular, *accept*, *connect*, *send*, *recv*, *read*, and *write* can all return EWOULDBLOCK, and processes should be prepared to deal with such return codes. If an operation such as a *send* cannot be done in its entirety, but partial writes are sensible (for example, when using a stream socket), the data that can be sent immediately will be processed, and the return value will indicate the amount actually sent.

5.3. Interrupt driven socket I/O

The SIGIO signal allows a process to be notified via a signal when a socket (or more generally, a file descriptor) has data waiting to be read. Use of the SIGIO facility requires three steps: First, the process must set up a SIGIO signal handler by use of the *signal* or *sigvec* calls. Second, it must set the process id or process group id which is to receive notification of pending input to its own process id, or the process group id of its process group (note that the default process group of a socket is group zero). This is accomplished by use of an *fcntl* call. Third, it must enable asynchronous notification of pending I/O requests with another *fcntl* call. Sample code to allow a given process to receive information on pending I/O requests as they occur for a socket *s* is given in Figure 6. With the addition of a handler for SIGURG, this code can also be used to prepare for receipt of SIGURG signals.

```
#include <fcntl.h>
 ...
int     io_handler();
 ...
signal(SIGIO, io_handler);

/* Set the process receiving SIGIO/SIGURG signals to us */

if (fcntl(s, F_SETOWN, getpid()) < 0) {
        perror("fcntl F_SETOWN");
        exit(1);
}

/* Allow receipt of asynchronous I/O signals */

if (fcntl(s, F_SETFL, FASYNC) < 0) {
        perror("fcntl F_SETFL, FASYNC");
        exit(1);
}
```

Figure 6. Use of asynchronous notification of I/O requests.

5.4. Signals and process groups

Due to the existence of the SIGURG and SIGIO signals each socket has an associated process number, just as is done for terminals. This value is initialized to zero, but may be redefined at a later time with the F_SETOWN *fcntl*, such as was done in the code above for SIGIO. To set the socket's process id for signals, positive arguments should be given to the *fcntl* call. To set the socket's process group for signals, negative arguments should be passed to *fcntl*. Note that the process number indicates either the associated process id or the associated process group; it is impossible to specify both at the same time. A similar *fcntl*, F_GETOWN, is available for determining the current process number of a socket.

Another signal which is useful when constructing server processes is SIGCHLD. This signal is delivered to a process when any child processes have changed state. Normally servers use the signal to "reap" child processes that have exited without explicitly awaiting their termination or periodic polling for exit status. For example, the remote login server loop shown in Figure 2 may be augmented as shown in Figure 7.

```
int reaper();
 ...
signal(SIGCHLD, reaper);
listen(f, 5);
for (;;) {
        int g, len = sizeof (from);

        g = accept(f, (struct sockaddr *)&from, &len,);
        if (g < 0) {
                if (errno != EINTR)
                        syslog(LOG_ERR, "rlogind: accept: %m");
                continue;
        }
        ...
}
 ...
#include <wait.h>
reaper()
{
        union wait status;

        while (wait3(&status, WNOHANG, 0) > 0)
                ;
}
```

Figure 7. Use of the SIGCHLD signal.

If the parent server process fails to reap its children, a large number of "zombie" processes may be created.

5.5. Pseudo terminals

Many programs will not function properly without a terminal for standard input and output. Since sockets do not provide the semantics of terminals, it is often necessary to have a process communicating over the network do so through a *pseudo-terminal*. A pseudo- terminal is actually a pair of devices, master and slave, which allow a process to serve as an active agent in communication between processes and users. Data written on the slave side of a pseudo-terminal is supplied as input to a process reading from the master side, while data written on the master side are processed as terminal input for the slave.

In this way, the process manipulating the master side of the pseudo-terminal has control over the information read and written on the slave side as if it were manipulating the keyboard and reading the screen on a real terminal. The purpose of this abstraction is to preserve terminal semantics over a network connection— that is, the slave side appears as a normal terminal to any process reading from or writing to it.

For example, the remote login server uses pseudo-terminals for remote login sessions. A user logging in to a machine across the network is provided a shell with a slave pseudo-terminal as standard input, output, and error. The server process then handles the communication between the programs invoked by the remote shell and the user's local client process. When a user sends a character that generates an interrupt on the remote machine that flushes terminal output, the pseudo-terminal generates a control message for the server process. The server then sends an out of band message to the client process to signal a flush of data at the real terminal and on the intervening data buffered in the network.

Under 4.4BSD, the name of the slave side of a pseudo-terminal is of the form */dev/ttyxy*, where *x* is a single letter starting at 'p' and continuing to 't'. *y* is a hexadecimal digit (i.e., a single character in the range 0 through 9 or 'a' through 'f'). The master side of a pseudo-terminal is */dev/ptyxy*, where *x* and *y* correspond to the slave side of the pseudo-terminal.

In general, the method of obtaining a pair of master and slave pseudo-terminals is to find a pseudo-terminal which is not currently in use. The master half of a pseudo-terminal is a single-open device; thus, each master may be opened in turn until an open succeeds. The slave side of the pseudo-terminal is then opened, and is set to the proper terminal modes if necessary. The process then *fork*s; the child closes the master side of the pseudo-terminal, and *exec*s the appropriate program. Meanwhile, the parent closes the slave side of the pseudo-terminal and begins reading and writing from the master side. Sample code making use of pseudo-terminals is given in Figure 8; this code assumes that a connection on a socket *s* exists, connected to a peer who wants a service of some kind, and that the process has disassociated itself from any previous controlling terminal.

5.6. Selecting specific protocols

If the third argument to the *socket* call is 0, *socket* will select a default protocol to use with the returned socket of the type requested. The default protocol is usually correct, and alternate choices are not usually available. However, when using ''raw'' sockets to communicate directly with lower-level protocols or hardware interfaces, the protocol argument may be important for setting up demultiplexing. For example, raw sockets in the Internet family may be used to implement a new protocol above IP, and the socket will receive packets only for the protocol specified. To obtain a particular protocol one determines the protocol number as defined within the communication domain. For the Internet domain one may use one of the library routines discussed in section 3, such as *getprotobyname*:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
 ...
pp = getprotobyname("newtcp");
s = socket(AF_INET, SOCK_STREAM, pp->p_proto);
```

This would result in a socket *s* using a stream based connection, but with protocol type of ''newtcp'' instead of the default ''tcp.''

In the NS domain, the available socket protocols are defined in <*netns/ns.h*>. To create a raw socket for Xerox Error Protocol messages, one might use:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
 ...
s = socket(AF_NS, SOCK_RAW, NSPROTO_ERROR);
```

```
gotpty = 0;
for (c = 'p'; !gotpty && c <= 's'; c++) {
        line = "/dev/ptyXX";
        line[sizeof("/dev/pty")-1] = c;
        line[sizeof("/dev/ptyp")-1] = '0';
        if (stat(line, &statbuf) < 0)
                break;
        for (i = 0; i < 16; i++) {
                line[sizeof("/dev/ptyp")-1] = "0123456789abcdef"[i];
                master = open(line, O_RDWR);
                if (master > 0) {
                        gotpty = 1;
                        break;
                }
        }
}
if (!gotpty) {
        syslog(LOG_ERR, "All network ports in use");
        exit(1);
}

line[sizeof("/dev/")-1] = 't';
slave = open(line, O_RDWR);/* slave is now slave side */
if (slave < 0) {
        syslog(LOG_ERR, "Cannot open slave pty %s", line);
        exit(1);
}

ioctl(slave, TIOCGETP, &b);  /* Set slave tty modes */
b.sg_flags = CRMOD|XTABS|ANYP;
ioctl(slave, TIOCSETP, &b);

i = fork();
if (i < 0) {
        syslog(LOG_ERR, "fork: %m");
        exit(1);
} else if (i) {                 /* Parent */
        close(slave);
        ...
} else {                /* Child */
        (void) close(s);
        (void) close(master);
        dup2(slave, 0);
        dup2(slave, 1);
        dup2(slave, 2);
        if (slave > 2)
                (void) close(slave);
        ...
}
```

Figure 8. Creation and use of a pseudo terminal

5.7. Address binding

As was mentioned in section 2, binding addresses to sockets in the Internet and NS domains can be fairly complex. As a brief reminder, these associations are composed of local and foreign addresses, and local and foreign ports. Port numbers are allocated out of separate spaces, one for each system and one for each domain on that system. Through the *bind* system call, a process may specify half of an association, the <local address, local port> part, while the *connect* and *accept* primitives are used to complete a socket's association by specifying the <foreign address, foreign port> part. Since the association is created in two steps the association uniqueness requirement indicated previously could be violated unless care is taken. Further, it is unrealistic to expect user programs to always know proper values to use for the local address and local port since a host may reside on multiple networks and the set of allocated port numbers is not directly accessible to a user.

To simplify local address binding in the Internet domain the notion of a ''wildcard'' address has been provided. When an address is specified as INADDR_ANY (a manifest constant defined in <netinet/in.h>), the system interprets the address as ''any valid address''. For example, to bind a specific port number to a socket, but leave the local address unspecified, the following code might be used:

```
#include <sys/types.h>
#include <netinet/in.h>
 ...
struct sockaddr_in sin;
 ...
s = socket(AF_INET, SOCK_STREAM, 0);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

Sockets with wildcarded local addresses may receive messages directed to the specified port number, and sent to any of the possible addresses assigned to a host. For example, if a host has addresses 128.32.0.4 and 10.0.0.78, and a socket is bound as above, the process will be able to accept connection requests which are addressed to 128.32.0.4 or 10.0.0.78. If a server process wished to only allow hosts on a given network connect to it, it would bind the address of the host on the appropriate network.

In a similar fashion, a local port may be left unspecified (specified as zero), in which case the system will select an appropriate port number for it. This shortcut will work both in the Internet and NS domains. For example, to bind a specific local address to a socket, but to leave the local port number unspecified:

```
hp = gethostbyname(hostname);
if (hp == NULL) {
        ...
}
bcopy(hp->h_addr, (char *) sin.sin_addr, hp->h_length);
sin.sin_port = htons(0);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

The system selects the local port number based on two criteria. The first is that on 4BSD systems, Internet ports below IPPORT_RESERVED (1024) (for the Xerox domain, 0 through 3000) are reserved for privileged users (i.e., the super user); Internet ports above IPPORT_USERRESERVED (50000) are reserved for non-privileged servers. The second is that the port number is not currently bound to some other socket. In order to find a free Internet port number in the privileged range the *rresvport* library routine may be used as follows to return a stream socket in with a privileged port number:

```
int lport = IPPORT_RESERVED – 1;
int s;
...
s = rresvport(&lport);
if (s < 0) {
        if (errno == EAGAIN)
                fprintf(stderr, "socket: all ports in use\n");
        else
                perror("rresvport: socket");
        ...
}
```

The restriction on allocating ports was done to allow processes executing in a ‘‘secure’’ environment to perform authentication based on the originating address and port number. For example, the *rlogin*(1) command allows users to log in across a network without being asked for a password, if two conditions hold: First, the name of the system the user is logging in from is in the file */etc/hosts.equiv* on the system he is logging in to (or the system name and the user name are in the user's *.rhosts* file in the user's home directory), and second, that the user's rlogin process is coming from a privileged port on the machine from which he is logging. The port number and network address of the machine from which the user is logging in can be determined either by the *from* result of the *accept* call, or from the *getpeername* call.

In certain cases the algorithm used by the system in selecting port numbers is unsuitable for an application. This is because associations are created in a two step process. For example, the Internet file transfer protocol, FTP, specifies that data connections must always originate from the same local port. However, duplicate associations are avoided by connecting to different foreign ports. In this situation the system would disallow binding the same local address and port number to a socket if a previous data connection's socket still existed. To override the default port selection algorithm, an option call must be performed prior to address binding:

```
 ...
int     on = 1;
 ...
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

With the above call, local addresses may be bound which are already in use. This does not violate the uniqueness requirement as the system still checks at connect time to be sure any other sockets with the same local address and port do not have the same foreign address and port. If the association already exists, the error EADDRINUSE is returned. A related socket option, SO_REUSEPORT, which allows completely duplicate bindings, is described in the IP multicasting section.

5.8. Socket Options

It is possible to set and get a number of options on sockets via the *setsockopt* and *getsockopt* system calls. These options include such things as marking a socket for broadcasting, not to route, to linger on close, etc. In addition, there are protocol-specific options for IP and TCP, as described in \$1\$2 \$1\$2 and in the section on multicasting below.

The general forms of the calls are:

```
setsockopt(s, level, optname, optval, optlen);
```

and

```
getsockopt(s, level, optname, optval, optlen);
```

The parameters to the calls are as follows: *s* is the socket on which the option is to be applied. *Level* specifies the protocol layer on which the option is to be applied; in most cases this is the ''socket level'', indicated by the symbolic constant SOL_SOCKET, defined in <*sys/socket.h*>. The actual option is specified in *optname*, and is a symbolic constant also defined in <*sys/socket.h*>. *Optval* and *Optlen* point to the value of the option (in most cases, whether the option is to be turned on or off), and the length of the value of the option, respectively. For *getsockopt*, *optlen* is a value-result parameter, initially set to the size of the storage area pointed to by *optval*, and modified upon return to indicate the actual amount of storage used.

An example should help clarify things. It is sometimes useful to determine the type (e.g., stream, datagram, etc.) of an existing socket; programs under *inetd* (described below) may need to perform this task. This can be accomplished as follows via the SO_TYPE socket option and the *getsockopt* call:

```
#include <sys/types.h>
#include <sys/socket.h>

int type, size;

size = sizeof (int);

if (getsockopt(s, SOL_SOCKET, SO_TYPE, (char *) &type, &size) < 0) {
        ...
}
```

After the *getsockopt* call, *type* will be set to the value of the socket type, as defined in <*sys/socket.h*>. If, for example, the socket were a datagram socket, *type* would have the value corresponding to SOCK_DGRAM.

5.9. Broadcasting and determining network configuration

By using a datagram socket, it is possible to send broadcast packets on many networks supported by the system. The network itself must support broadcast; the system provides no simulation of broadcast in software. Broadcast messages can place a high load on a network since they force every host on the network to service them. Consequently, the ability to send broadcast packets has been limited to sockets which are explicitly marked as allowing broadcasting. Broadcast is typically used for one of two reasons: it is desired to find a resource on a local network without prior knowledge of its address, or important functions such as routing require that information be sent to all accessible neighbors.

Multicasting is an alternative to broadcasting. Setting up IP multicast sockets is described in the next section.

To send a broadcast message, a datagram socket should be created:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

or

```
s = socket(AF_NS, SOCK_DGRAM, 0);
```

The socket is marked as allowing broadcasting,

```
int     on = 1;

setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof (on));
```

and at least a port number should be bound to the socket:

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

or, for the NS domain,

```
sns.sns_family = AF_NS;
netnum = htonl(net);
sns.sns_addr.x_net = *(union ns_net *) &netnum; /* insert net number */
sns.sns_addr.x_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sns, sizeof (sns));
```

The destination address of the message to be broadcast depends on the network(s) on which the message is to be broadcast. The Internet domain supports a shorthand notation for broadcast on the local network, the address INADDR_BROADCAST (defined in *<netinet/in.h>*. To determine the list of addresses for all reachable neighbors requires knowledge of the networks to which the host is connected. Since this information should be obtained in a host-independent fashion and may be impossible to derive, 4.4BSD provides a method of retrieving this information from the system data structures. The SIOCGIFCONF *ioctl* call returns the interface configuration of a host in the form of a single *ifconf* structure; this structure contains a ‘‘data area’’ which is made up of an array of of *ifreq* structures, one for each network interface to which the host is connected. These structures are defined in *<net/if.h>* as follows:

```
struct ifconf {
        int     ifc_len;                        /* size of associated buffer */
        union {
                caddr_t  ifcu_buf;
                struct   ifreq *ifcu_req;
        } ifc_ifcu;
};

#define  ifc_buf  ifc_ifcu.ifcu_buf            /* buffer address */
#define  ifc_req  ifc_ifcu.ifcu_req            /* array of structures returned */

#define  IFNAMSIZ       16

struct ifreq {
        char    ifr_name[IFNAMSIZ];             /* if name, e.g. "en0" */
        union {
                struct   sockaddr ifru_addr;
                struct   sockaddr ifru_dstaddr;
                struct   sockaddr ifru_broadaddr;
                short    ifru_flags;
                caddr_t  ifru_data;
        } ifr_ifru;
};

#define  ifr_addr        ifr_ifru.ifru_addr      /* address */
#define  ifr_dstaddr     ifr_ifru.ifru_dstaddr   /* other end of p-to-p link */
#define  ifr_broadaddr   ifr_ifru.ifru_broadaddr /* broadcast address */
#define  ifr_flags       ifr_ifru.ifru_flags     /* flags */
#define  ifr_data        ifr_ifru.ifru_data      /* for use by interface */
```

The actual call which obtains the interface configuration is

```
struct ifconf ifc;
char buf[BUFSIZ];

ifc.ifc_len = sizeof (buf);
ifc.ifc_buf = buf;
if (ioctl(s, SIOCGIFCONF, (char *) &ifc) < 0) {
        ...
}
```

After this call *buf* will contain one *ifreq* structure for each network to which the host is connected, and *ifc.ifc_len* will have been modified to reflect the number of bytes used by the *ifreq* structures.

For each structure there exists a set of ‘‘interface flags’’ which tell whether the network corresponding to that interface is up or down, point to point or broadcast, etc. The SIOCGIFFLAGS *ioctl* retrieves these flags for an interface specified by an *ifreq* structure as follows:

```
struct ifreq *ifr;

ifr = ifc.ifc_req;

for (n = ifc.ifc_len / sizeof (struct ifreq); --n >= 0; ifr++) {
        /*
         * We must be careful that we don't use an interface
         * devoted to an address family other than those intended;
         * if we were interested in NS interfaces, the
         * AF_INET would be AF_NS.
         */
        if (ifr->ifr_addr.sa_family != AF_INET)
                continue;
        if (ioctl(s, SIOCGIFFLAGS, (char *) ifr) < 0) {
                ...
        }
        /*
         * Skip boring cases.
         */
        if ((ifr->ifr_flags & IFF_UP) == 0 ||
            (ifr->ifr_flags & IFF_LOOPBACK) ||
            (ifr->ifr_flags & (IFF_BROADCAST | IFF_POINTTOPOINT)) == 0)
                continue;
```

Once the flags have been obtained, the broadcast address must be obtained. In the case of broadcast networks this is done via the SIOCGIFBRDADDR *ioctl*, while for point-to-point networks the address of the destination host is obtained with SIOCGIFDSTADDR.

```
struct sockaddr dst;

if (ifr->ifr_flags & IFF_POINTTOPOINT) {
        if (ioctl(s, SIOCGIFDSTADDR, (char *) ifr) < 0) {
                ...
        }
        bcopy((char *) ifr->ifr_dstaddr, (char *) &dst, sizeof (ifr->ifr_dstaddr));
} else if (ifr->ifr_flags & IFF_BROADCAST) {
        if (ioctl(s, SIOCGIFBRDADDR, (char *) ifr) < 0) {
                ...
        }
        bcopy((char *) ifr->ifr_broadaddr, (char *) &dst, sizeof (ifr->ifr_broadaddr));
}
```

After the appropriate *ioctl*'s have obtained the broadcast or destination address (now in *dst*), the *sendto* call may be used:

```
        sendto(s, buf, buflen, 0, (struct sockaddr *)&dst, sizeof (dst));
}
```

In the above loop one *sendto* occurs for every interface to which the host is connected that supports the notion of broadcast or point-to-point addressing. If a process only wished to send broadcast messages on a given network, code similar to that outlined above would be used, but the loop would need to find the correct destination address.

Received broadcast messages contain the senders address and port, as datagram sockets are bound before a message is allowed to go out.

5.10. IP Multicasting

IP multicasting is the transmission of an IP datagram to a "host group", a set of zero or more hosts identified by a single IP destination address. A multicast datagram is delivered to all members of its destination host group with the same "best-efforts" reliability as regular unicast IP datagrams, i.e., the datagram is not guaranteed to arrive intact at all members of the destination group or in the same order relative to other datagrams.

The membership of a host group is dynamic; that is, hosts may join and leave groups at any time. There is no restriction on the location or number of members in a host group. A host may be a member of more than one group at a time. A host need not be a member of a group to send datagrams to it.

A host group may be permanent or transient. A permanent group has a well-known, administratively assigned IP address. It is the address, not the membership of the group, that is permanent; at any time a permanent group may have any number of members, even zero. Those IP multicast addresses that are not reserved for permanent groups are available for dynamic assignment to transient groups which exist only as long as they have members.

In general, a host cannot assume that datagrams sent to any host group address will reach only the intended hosts, or that datagrams received as a member of a transient host group are intended for the recipient. Misdelivery must be detected at a level above IP, using higher-level identifiers or authentication tokens. Information transmitted to a host group address should be encrypted or governed by administrative routing controls if the sender is concerned about unwanted listeners.

IP multicasting is currently supported only on AF_INET sockets of type SOCK_DGRAM and SOCK_RAW, and only on subnetworks for which the interface driver has been modified to support multicasting.

The next subsections describe how to send and receive multicast datagrams.

5.10.1. Sending IP Multicast Datagrams

To send a multicast datagram, specify an IP multicast address in the range 224.0.0.0 to 239.255.255.255 as the destination address in a \$1\$2 call.

The definitions required for the multicast-related socket options are found in *<netinet/in.h>*. All IP addresses are passed in network byte-order.

By default, IP multicast datagrams are sent with a time-to-live (TTL) of 1, which prevents them from being forwarded beyond a single subnetwork. A new socket option allows the TTL for subsequent multicast datagrams to be set to any value from 0 to 255, in order to control the scope of the multicasts:

```
u_char ttl;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, &ttl, sizeof(ttl));
```

Multicast datagrams with a TTL of 0 will not be transmitted on any subnet, but may be delivered locally if the sending host belongs to the destination group and if multicast loopback has not been disabled on the sending socket (see below). Multicast datagrams with TTL greater than one may be delivered to more than one subnet if there are one or more multicast routers attached to the first-hop subnet. To provide meaningful scope control, the multicast routers support the notion of TTL "thresholds", which prevent datagrams with less than a certain TTL from traversing certain subnets. The thresholds enforce the following convention:

Scope	Initial TTL
restricted to the same host	0
restricted to the same subnet	1
restricted to the same site	32
restricted to the same region	64
restricted to the same continent	128
unrestricted	255

"Sites" and "regions" are not strictly defined, and sites may be further subdivided into smaller administrative units, as a local matter.

An application may choose an initial TTL other than the ones listed above. For example, an application might perform an "expanding-ring search" for a network resource by sending a multicast query, first with a TTL of 0, and then with larger and larger TTLs, until a reply is received, perhaps using the TTL sequence 0, 1, 2, 4, 8, 16, 32.

The multicast router \$1\$2 refuses to forward any multicast datagram with a destination address between 224.0.0.0 and 224.0.0.255, inclusive, regardless of its TTL. This range of addresses is reserved for the use of routing protocols and other low-level topology discovery or maintenance protocols, such as gateway discovery and group membership reporting.

The address 224.0.0.0 is guaranteed not to be assigned to any group, and 224.0.0.1 is assigned to the permanent group of all IP hosts (including gateways). This is used to address all multicast hosts on the directly connected network. There is no multicast address (or any other IP address) for all hosts on the total Internet. The addresses of other well-known, permanent groups are published in the "Assigned Numbers" RFC, which is available from the InterNIC.

Each multicast transmission is sent from a single network interface, even if the host has more than one multicast-capable interface. (If the host is also serving as a multicast router, a multicast may be *forwarded* to interfaces other than originating interface, provided that the TTL is greater than 1.) The default interface to be used for multicasting is the primary network interface on the system. A socket option is available to override the default for subsequent transmissions from a given socket:

```
struct in_addr addr;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_IF, &addr, sizeof(addr));
```

where "addr" is the local IP address of the desired outgoing interface. An address of INADDR_ANY may be used to revert to the default interface. The local IP address of an interface can be obtained via the SIOCGIFCONF ioctl. To determine if an interface supports multicasting, fetch the interface flags via the SIOCGIFFLAGS ioctl and see if the IFF_MULTICAST flag is set. (Normal applications should not need to use this option; it is intended primarily for multicast routers and other system services specifically concerned with internet topology.) The SIOCGIFCONF and SIOCGIFFLAGS ioctls are described in the previous section.

If a multicast datagram is sent to a group to which the sending host itself belongs (on the outgoing interface), a copy of the datagram is, by default, looped back by the IP layer for local delivery. Another socket option gives the sender explicit control over whether or not subsequent datagrams are looped back:

```
u_char loop;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_LOOP, &loop, sizeof(loop));
```

where *loop* is set to 0 to disable loopback, and set to 1 to enable loopback. This option improves performance for applications that may have no more than one instance on a single host (such as a router demon), by eliminating the overhead of receiving their own transmissions. It should generally not be used by applications for which there may be more than one instance on a single host (such as a conferencing program) or for which the sender does not belong to the destination group (such as a time querying program).

A multicast datagram sent with an initial TTL greater than 1 may be delivered to the sending host on a different interface from that on which it was sent, if the host belongs to the destination group on that other interface. The loopback control option has no effect on such delivery.

5.10.2. Receiving IP Multicast Datagrams

Before a host can receive IP multicast datagrams, it must become a member of one or more IP multicast groups. A process can ask the host to join a multicast group by using the following socket option:

```
struct ip_mreq mreq;
setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq, sizeof(mreq))
```

where "mreq" is the following structure:

```
struct ip_mreq {
    struct in_addr imr_multiaddr; /* multicast group to join */
    struct in_addr imr_interface; /* interface to join on */
}
```

Every membership is associated with a single interface, and it is possible to join the same group on more than one interface. "imr_interface" should be INADDR_ANY to choose the default multicast interface, or one of the host's local addresses to choose a particular (multicast-capable) interface. Up to IP_MAX_MEMBERSHIPS (currently 20) memberships may be added on a single socket.

To drop a membership, use:

```
struct ip_mreq mreq;
setsockopt(sock, IPPROTO_IP, IP_DROP_MEMBERSHIP, &mreq, sizeof(mreq));
```

where "mreq" contains the same values as used to add the membership. The memberships associated with a socket are also dropped when the socket is closed or the process holding the socket is killed. However, more than one socket may claim a membership in a particular group, and the host will remain a member of that group until the last claim is dropped.

The memberships associated with a socket do not necessarily determine which datagrams are received on that socket. Incoming multicast packets are accepted by the kernel IP layer if any socket has claimed a membership in the destination group of the datagram; however, delivery of a multicast

datagram to a particular socket is based on the destination port (or protocol type, for raw sockets), just as with unicast datagrams. To receive multicast datagrams sent to a particular port, it is necessary to bind to that local port, leaving the local address unspecified (i.e., INADDR_ANY). To receive multicast datagrams sent to a particular group and port, bind to the local port, with the local address set to the multicast group address. Once bound to a multicast address, the socket cannot be used for sending data.

More than one process may bind to the same SOCK_DGRAM UDP port or the same multicast group and port if the *bind* call is preceded by:

```
int on = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEPORT, &on, sizeof(on));
```

All processes sharing the port must enable this option. Every incoming multicast or broadcast UDP datagram destined to the shared port is delivered to all sockets bound to the port. For backwards compatibility reasons, this does not apply to incoming unicast datagrams. Unicast datagrams are never delivered to more than one socket, regardless of how many sockets are bound to the datagram's destination port.

A final multicast-related extension is independent of IP: two new ioctls, SIOCADDMULTI and SIOCDELMULTI, are available to add or delete link-level (e.g., Ethernet) multicast addresses accepted by a particular interface. The address to be added or deleted is passed as a sockaddr structure of family AF_UNSPEC, within the standard ifreq structure.

These ioctls are for the use of protocols other than IP, and require superuser privileges. A link-level multicast address added via SIOCADDMULTI is not automatically deleted when the socket used to add it goes away; it must be explicitly deleted. It is inadvisable to delete a link-level address that may be in use by IP.

5.10.3. Sample Multicast Program

The following program sends or receives multicast packets. If invoked with one argument, it sends a packet containing the current time to an arbitrarily-chosen multicast group and UDP port. If invoked with no arguments, it receives and prints these packets. Start it as a sender on just one host and as a receiver on all the other hosts.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <time.h>
#include <stdio.h>

#define EXAMPLE_PORT    60123
#define EXAMPLE_GROUP   "224.0.0.250"

main(argc)
    int argc;
{
    struct sockaddr_in addr;
    int addrlen, fd, cnt;
    struct ip_mreq mreq;
    char message[50];

    fd = socket(AF_INET, SOCK_DGRAM, 0);
    if (fd < 0) {
        perror("socket");
        exit(1);
    }
```

```
        bzero(&addr, sizeof(addr));
        addr.sin_family = AF_INET;
        addr.sin_addr.s_addr = htonl(INADDR_ANY);
        addr.sin_port = htons(EXAMPLE_PORT);
        addrlen = sizeof(addr);
        if (argc > 1) {      /* Send */
            addr.sin_addr.s_addr = inet_addr(EXAMPLE_GROUP);
            while (1) {
                time_t t = time(0);
                sprintf(message, "time is %-24.24s", ctime(&t));
                cnt = sendto(fd, message, sizeof(message), 0,
                        (struct sockaddr *)&addr, addrlen);
                if (cnt < 0) {
                    perror("sendto");
                    exit(1);
                }
                sleep(5);
            }
        } else {             /* Receive */
            if (bind(fd, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
                perror("bind");
                exit(1);
            }

            mreq.imr_multiaddr.s_addr = inet_addr(EXAMPLE_GROUP);
            mreq.imr_interface.s_addr = htonl(INADDR_ANY);
            if (setsockopt(fd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                        &mreq, sizeof(mreq)) < 0) {
                perror("setsockopt mreq");
                exit(1);
            }

            while (1) {
                cnt = recvfrom(fd, message, sizeof(message), 0,
                            (struct sockaddr *)&addr, &addrlen);
                if (cnt <= 0) {
                        if (cnt == 0) {
                            break;
                        }
                        perror("recvfrom");
                        exit(1);
                }
                printf("%s: message = \"%s\"\n",
                        inet_ntoa(addr.sin_addr), message);
            }
        }
    }
```

5.11. NS Packet Sequences

The semantics of NS connections demand that the user both be able to look inside the network header associated with any incoming packet and be able to specify what should go in certain fields of an outgoing packet. Using different calls to *setsockopt*, it is possible to indicate whether prototype headers

will be associated by the user with each outgoing packet (SO_HEADERS_ON_OUTPUT), to indicate whether the headers received by the system should be delivered to the user (SO_HEADERS_ON_INPUT), or to indicate default information that should be associated with all outgoing packets on a given socket (SO_DEFAULT_HEADERS).

The contents of a SPP header (minus the IDP header) are:

```
struct sphdr {
        u_char  sp_cc;                  /* connection control */
#define SP_SP   0x80                    /* system packet */
#define SP_SA   0x40                    /* send acknowledgement */
#define SP_OB   0x20                    /* attention (out of band data) */
#define SP_EM   0x10                    /* end of message */

        u_char  sp_dt;                  /* datastream type */
        u_short sp_sid;                 /* source connection identifier */
        u_short sp_did;                 /* destination connection identifier */
        u_short sp_seq;                 /* sequence number */
        u_short sp_ack;                 /* acknowledge number */
        u_short sp_alo;                 /* allocation number */
};
```

Here, the items of interest are the *datastream type* and the *connection control* fields. The semantics of the datastream type are defined by the application(s) in question; the value of this field is, by default, zero, but it can be used to indicate things such as Xerox's Bulk Data Transfer Protocol (in which case it is set to one). The connection control field is a mask of the flags defined just below it. The user may set or clear the end-of-message bit to indicate that a given message is the last of a given substream type, or may set/clear the attention bit as an alternate way to indicate that a packet should be sent out-of-band. As an example, to associate prototype headers with outgoing SPP packets, consider:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/sp.h>
 ...
struct sockaddr_ns sns, to;
int s, on = 1;
struct databuf {
        struct sphdr proto_spp;  /* prototype header */
        char buf[534];           /* max. possible data by Xerox std. */
} buf;
 ...
s = socket(AF_NS, SOCK_SEQPACKET, 0);
 ...
bind(s, (struct sockaddr *) &sns, sizeof (sns));
setsockopt(s, NSPROTO_SPP, SO_HEADERS_ON_OUTPUT, &on, sizeof(on));
 ...
buf.proto_spp.sp_dt = 1;/* bulk data */
buf.proto_spp.sp_cc = SP_EM;        /* end-of-message */
strcpy(buf.buf, "hello world\n");
sendto(s, (char *) &buf, sizeof(struct sphdr) + strlen("hello world\n"),
    (struct sockaddr *) &to, sizeof(to));
 ...
```

Note that one must be careful when writing headers; if the prototype header is not written with the data with which it is to be associated, the kernel will treat the first few bytes of the data as the header, with

unpredictable results. To turn off the above association, and to indicate that packet headers received by the system should be passed up to the user, one might use:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/sp.h>
 ...
struct sockaddr sns;
int s, on = 1, off = 0;
 ...
s = socket(AF_NS, SOCK_SEQPACKET, 0);
 ...
bind(s, (struct sockaddr *) &sns, sizeof (sns));
setsockopt(s, NSPROTO_SPP, SO_HEADERS_ON_OUTPUT, &off, sizeof(off));
setsockopt(s, NSPROTO_SPP, SO_HEADERS_ON_INPUT, &on, sizeof(on));
```

Output is handled somewhat differently in the IDP world. The header of an IDP-level packet looks like:

```
struct idp {
        u_short         idp_sum;                /* Checksum */
        u_short         idp_len;                /* Length, in bytes, including header */
        u_char          idp_tc;                 /* Transport Control (i.e., hop count) */
        u_char          idp_pt;                 /* Packet Type (i.e., level 2 protocol) */
        struct ns_addr  idp_dna;                /* Destination Network Address */
        struct ns_addr  idp_sna;                /* Source Network Address */
};
```

The primary field of interest in an IDP header is the *packet type* field. The standard values for this field are (as defined in <*netns/ns.h*>):

```
#define NSPROTO_RI      1       /* Routing Information */
#define NSPROTO_ECHO    2       /* Echo Protocol */
#define NSPROTO_ERROR   3       /* Error Protocol */
#define NSPROTO_PE      4       /* Packet Exchange */
#define NSPROTO_SPP     5       /* Sequenced Packet */
```

For SPP connections, the contents of this field are automatically set to NSPROTO_SPP; for IDP packets, this value defaults to zero, which means ‘‘unknown’’.

Setting the value of that field with SO_DEFAULT_HEADERS is easy:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/idp.h>
 ...
struct sockaddr sns;
struct idp proto_idp;           /* prototype header */
int s, on = 1;
 ...
```

```
s = socket(AF_NS, SOCK_DGRAM, 0);
 ...
bind(s, (struct sockaddr *) &sns, sizeof (sns));
proto_idp.idp_pt = NSPROTO_PE;  /* packet exchange */
setsockopt(s, NSPROTO_IDP, SO_DEFAULT_HEADERS, (char *) &proto_idp,
    sizeof(proto_idp));
 ...
```

Using SO_HEADERS_ON_OUTPUT is somewhat more difficult. When SO_HEADERS_ON_OUTPUT is turned on for an IDP socket, the socket becomes (for all intents and purposes) a raw socket. In this case, all the fields of the prototype header (except the length and checksum fields, which are computed by the kernel) must be filled in correctly in order for the socket to send and receive data in a sensible manner. To be more specific, the source address must be set to that of the host sending the data; the destination address must be set to that of the host for whom the data is intended; the packet type must be set to whatever value is desired; and the hopcount must be set to some reasonable value (almost always zero). It should also be noted that simply sending data using *write* will not work unless a *connect* or *sendto* call is used, in spite of the fact that it is the destination address in the prototype header that is used, not the one given in either of those calls. For almost all IDP applications , using SO_DEFAULT_HEADERS is easier and more desirable than writing headers.

5.12. Three-way Handshake

The semantics of SPP connections indicates that a three-way handshake, involving changes in the datastream type, should — but is not absolutely required to — take place before a SPP connection is closed. Almost all SPP connections are ‘‘well-behaved’’ in this manner; when communicating with any process, it is best to assume that the three-way handshake is required unless it is known for certain that it is not required. In a three-way close, the closing process indicates that it wishes to close the connection by sending a zero-length packet with end-of-message set and with datastream type 254. The other side of the connection indicates that it is OK to close by sending a zero-length packet with end-of-message set and datastream type 255. Finally, the closing process replies with a zero-length packet with substream type 255; at this point, the connection is considered closed. The following code fragments are simplified examples of how one might handle this three-way handshake at the user level; in the future, support for this type of close will probably be provided as part of the C library or as part of the kernel. The first code fragment below illustrates how a process might handle three-way handshake if it sees that the process it is communicating with wants to close the connection:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/sp.h>
 ...
#ifndef SPPSST_END
#define SPPSST_END 254
#define SPPSST_ENDREPLY 255
#endif
struct sphdr proto_sp;
int s;
 ...
```

```
read(s, buf, BUFSIZE);
if (((struct sphdr *)buf)->sp_dt == SPPSST_END) {
        /*
         * SPPSST_END indicates that the other side wants to
         * close.
         */
        proto_sp.sp_dt = SPPSST_ENDREPLY;
        proto_sp.sp_cc = SP_EM;
        setsockopt(s, NSPROTO_SPP, SO_DEFAULT_HEADERS, (char *)&proto_sp,
            sizeof(proto_sp));
        write(s, buf, 0);
        /*
         * Write a zero-length packet with datastream type = SPPSST_ENDREPLY
         * to indicate that the close is OK with us.  The packet that we
         * don't see (because we don't look for it) is another packet
         * from the other side of the connection, with SPPSST_ENDREPLY
         * on it it, too.  Once that packet is sent, the connection is
         * considered closed; note that we really ought to retransmit
         * the close for some time if we do not get a reply.
         */
        close(s);
}
 ...
```

To indicate to another process that we would like to close the connection, the following code would suffice:

```
#include <sys/types.h>
#include <sys/socket.h>
```

```
#include <netns/ns.h>
#include <netns/sp.h>
 ...
#ifndef SPPSST_END
#define SPPSST_END 254
#define SPPSST_ENDREPLY 255
#endif
struct sphdr proto_sp;
int s;
 ...
proto_sp.sp_dt = SPPSST_END;
proto_sp.sp_cc = SP_EM;
setsockopt(s, NSPROTO_SPP, SO_DEFAULT_HEADERS, (char *)&proto_sp,
    sizeof(proto_sp));
write(s, buf, 0);    /* send the end request */
proto_sp.sp_dt = SPPSST_ENDREPLY;
setsockopt(s, NSPROTO_SPP, SO_DEFAULT_HEADERS, (char *)&proto_sp,
    sizeof(proto_sp));
/*
 * We assume (perhaps unwisely) that the other side will send the ENDREPLY,
 * so we'll just send our final ENDREPLY as if we'd seen theirs already
 */
write(s, buf, 0);
close(s);
 ...
```

5.13. Packet Exchange

The Xerox standard protocols include a protocol that is both reliable and datagram-oriented. This protocol is known as Packet Exchange (PEX or PE) and, like SPP, is layered on top of IDP. PEX is important for a number of things: Courier remote procedure calls may be expedited through the use of PEX, and many Xerox servers are located by doing a PEX ''BroadcastForServers'' operation. Although there is no implementation of PEX in the kernel, it may be simulated at the user level with some clever coding and the use of one peculiar *getsockopt*. A PEX packet looks like:

```
/*
 * The packet-exchange header shown here is not defined
 * as part of any of the system include files.
 */
struct pex {
        struct idp  p_idp;                  /* idp header */
        u_short     ph_id[2];               /* unique transaction ID for pex */
        u_short     ph_client;              /* client type field for pex */
};
```

The *ph_id* field is used to hold a ''unique id'' that is used in duplicate suppression; the *ph_client* field indicates the PEX client type (similar to the packet type field in the IDP header). PEX reliability stems from the fact that it is an idempotent (''I send a packet to you, you send a packet to me'') protocol. Processes on each side of the connection may use the unique id to determine if they have seen a given packet before (the unique id field differs on each packet sent) so that duplicates may be detected, and to indicate which message a given packet is in response to. If a packet with a given unique id is sent and no response is received in a given amount of time, the packet is retransmitted until it is decided that no response will ever be received. To simulate PEX, one must be able to generate unique ids -- something that is hard to do at the user level with any real guarantee that the id is really unique. Therefore, a means (via *getsockopt*) has been provided for getting unique ids from the kernel. The following code fragment indicates how to get a unique id:

```
long uniqueid;
int s, idsize = sizeof(uniqueid);
 ...
s = socket(AF_NS, SOCK_DGRAM, 0);
 ...
/* get id from the kernel -- only on IDP sockets */
getsockopt(s, NSPROTO_PE, SO_SEQNO, (char *)&uniqueid, &idsize);
 ...
```

The retransmission and duplicate suppression code required to simulate PEX fully is left as an exercise for the reader.

5.14. Inetd

One of the daemons provided with 4.4BSD is *inetd*, the so called ‘‘internet super-server.’’ Having one daemon listen for requests for many daemons instead of having each daemon listen for its own requests reduces the number of idle daemons and simplies their implementation. *Inetd* handles two types of services: standard and TCPMUX. A standard service has a well-known port assigned to it and is listed in */etc/services* (see *services*(5)); it may be a service that implements an official Internet standard or is a BSD-specific service. TCPMUX services are nonstandard and do not have a well-known port assigned to them. They are invoked from *inetd* when a program connects to the "tcpmux" well-known port and specifies the service name. This is useful for adding locally-developed servers.

Inetd is invoked at boot time, and determines from the file */etc/inetd.conf* the servers for which it is to listen. Once this information has been read and a pristine environment created, *inetd* proceeds to create one socket for each service it is to listen for, binding the appropriate port number to each socket.

Inetd then performs a *select* on all these sockets for read availability, waiting for somebody wishing a connection to the service corresponding to that socket. *Inetd* then performs an *accept* on the socket in question, *fork*s, *dup*s the new socket to file descriptors 0 and 1 (stdin and stdout), closes other open file descriptors, and *exec*s the appropriate server.

Servers making use of *inetd* are considerably simplified, as *inetd* takes care of the majority of the IPC work required in establishing a connection. The server invoked by *inetd* expects the socket connected to its client on file descriptors 0 and 1, and may immediately perform any operations such as *read*, *write*, *send*, or *recv*. Indeed, servers may use buffered I/O as provided by the ‘‘stdio’’ conventions, as long as as they remember to use *fflush* when appropriate.

One call which may be of interest to individuals writing servers under *inetd* is the *getpeername* call, which returns the address of the peer (process) connected on the other end of the socket. For example, to log the Internet address in ‘‘dot notation’’ (e.g., ‘‘128.32.0.4’’) of a client connected to a server under *inetd*, the following code might be used:

```
struct sockaddr_in name;
int namelen = sizeof (name);
 ...
if (getpeername(0, (struct sockaddr *)&name, &namelen) < 0) {
        syslog(LOG_ERR, "getpeername: %m");
        exit(1);
} else
        syslog(LOG_INFO, "Connection from %s", inet_ntoa(name.sin_addr));
 ...
```

While the *getpeername* call is especially useful when writing programs to run with *inetd*, it can be used under other circumstances. Be warned, however, that *getpeername* will fail on UNIX domain sockets.

Standard TCP services are assigned unique well-known port numbers in the range of 0 to 1023 by the Internet Assigned Numbers Authority (IANA@ISI.EDU). The limited number of ports in this range are assigned to official Internet protocols. The TCPMUX service allows you to add locally-developed protocols without needing an official TCP port assignment. The TCPMUX protocol described in RFC-1078 is simple:

> "A TCP client connects to a foreign host on TCP port 1. It sends the service name followed by a carriage-return line-feed <CRLF>. The service name is never case sensitive. The server replies with a single character indicating positive ("+") or negative ("–") acknowledgment, immediately followed by an optional message of explanation, terminated with a <CRLF>. If the reply was positive, the selected protocol begins; otherwise the connection is closed."

In 4.4BSD, the TCPMUX service is built into \$*I*\$2 that is, \$*I*\$2 listens on TCP port 1 for requests for TCPMUX services listed in *inetd.conf*. \$*I*\$2 describes the format of TCPMUX entries for *inetd.conf*.

The following is an example TCPMUX server and its *inetd.conf* entry. More sophisticated servers may want to do additional processing before returning the positive or negative acknowledgement.

```
#include <sys/types.h>
#include <stdio.h>

main()
{
        time_t t;

        printf("+Go\r\n");
        fflush(stdout);
        time(&t);
        printf("%d = %s", t, ctime(&t));
        fflush(stdout);
}
```

The *inetd.conf* entry is:

```
tcpmux/current_time stream tcp nowait nobody /d/curtime curtime
```

Here's the portion of the client code that handles the TCPMUX handshake:

```
char line[BUFSIZ];
FILE *fp;
 ...

/* Use stdio for reading data from the server */
fp = fdopen(sock, "r");
if (fp == NULL) {
    fprintf(stderr, "Can't create file pointer\n");
    exit(1);
}

/* Send service request */
sprintf(line, "%s\r\n", "current_time");
if (write(sock, line, strlen(line)) < 0) {
    perror("write");
    exit(1);
}

/* Get ACK/NAK response from the server */
if (fgets(line, sizeof(line), fp) == NULL) {
    if (feof(fp)) {
        die();
    } else {
        fprintf(stderr, "Error reading response\n");
        exit(1);
    }
}

/* Delete <CR> */   ')) != NULL) {
if ((lp = index(line, '
    *lp = '  ';
}

switch (line[0]) {
    case '+':
            printf("Got ACK: %s\n", &line[1]);
            break;
    case '-':
            printf("Got NAK: %s\n", &line[1]);
            exit(0);
    default:
            printf("Got unknown response: %s\n", line);
            exit(1);
}

/* Get rest of data from the server */
while ((fgets(line, sizeof(line), fp)) != NULL) {
    fputs(line, stdout);
}
```