ICON/UXB Operating System Reference Manual

Volume 3

ICON
INTERNATIONAL

P.O. Box 340
Orem, Utah 84059
(801) 225-6888

OPERATING SYSTEM REFERENCE MANUAL

ICON/UXB

Supplementary Documents

Volume 3

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

This manual was prepared by the Documentation Group of Icon International, Inc., P.O. Box 340, Orem, UT 84057-0340. A form for reader's comments has been provided at the back of this publication. Comments are welcomed and may be sent to the above address. Users who respond will be entitled to free updates of this manual for one year.

Revision B

Order Number 172-022-004 (Manual Assembly)
Order Number 171-070-004 (Pages Only)

Printed in the U.S.A.

Change Record Page

Manual Part No. 172-022-004

Date	Revision	Description	Pages Affected
Jan. 1987	A	Initial production release	All
Nov. 1987	B	Incorporate additions of new supplementary documentation included in Releases 2.16, 3.0, and 3.1 of the ICON/UXB Operating System	Main cover, titlepage, Table of Contents, additions of the following documents: "PROC286 Software Support and the Dosc Command under ICON/UXB", "An Introduction to the Revision Control System", "Technical Note on ICON/UXB Magnetic Tape Support"

ICON/UXB Operating System Reference Manual

Volume 3 – Supplementary Documents

Icon International, Inc.

October, 1987

This volume contains documents which supplement the information in Volume 1 of the *ICON/UXB Operating System Reference Manual,* for the ICON version of the UNIX® operating system as distributed by U.C. Berkeley. The documents within this volume are grouped into the areas of system administration, languages, and supporting tools. This manual is a logical extension of Volume 2 of the *ICON/UXB Operating System Reference Manual,*

System Administration

40. 4.2BSD System Manual. W.N. Joy, E. cooper, R.S. Fabry, S.J. Leffler, M.K. McKusick, and D. Mosher.
    A concise, though terse, description of the system call interface provided in 4.2BSD. This will never be a best seller.
41. Fsck – The UNIX File System Check Program. M.K. McKusick and T.J. Kowalski.
    A reference document for use with the *fsck* program during times of file system distress.
42. 4.2BSD Line Printer Spooler Manual. R. Campbell.
    This document describes the structure and installation procedure for the line printer spooling system.
43. A Fast File System for UNIX. M.K. McKusick, W.N. Joy, S.J. Leffler, and R.S. Fabry.
    A description of the new file system organization design and implementation.
44. 4.2BSD Networking Implementation Notes. S.J. Leffler, W.N. Joy, and R.S. Fabry.
    A concise description of the system interfaces used within the networking subsystem.
45. Disc Quotas in a UNIX Environment. R. Elz.
    A light introduction to the care and feeding of the facilities which can be used in limiting disc resources.
46. Sendmail Installation and Operation Guide. E. Allman.
    The last word in installing and operating the *sendmail* program.
47. Sendmail — An Internetwork Mail Router. E. Allman.
    An overview document on the design and implementation of *sendmail.*

48. UNIX Implementation. K. Thompson.
    How the system actually works inside.

49. The UNIX I/O System. D.M. Ritchie.
    How the I/O system really works.

50. On the Security of UNIX. D.M. Ritchie.
    Hints on how to break the UNIX® operating system and how to avoid doing so.

51. Password Security: A Case History. R.H. Morris and K. Thompson.
    How the bad guys used to be able to break the password algorithm, and why they can't now, at least not so easily.

52. A Dial-Up Network of UNIX Systems. D.A. Nowitz and M.E. Lesk.
    Describes UUCP, a program for communicating files between computer systems using the UNIX® operating system.

53. UUCP Implementation Description. D.A. Nowitz.
    How UUCP works, and how to administer it.

Languages

54. The "C" Programming Language — Reference Manual. D.M. Ritchie.
    Official statement of the syntax and semantics of "C". Should be supplemented by *The C Programming Language,* B.W. Kernighan and D.M. Ritchie, Prentice–Hall, 1978, which contains a tutorial introduction and many examples.

55. Lint, A "C" Program Checker. S.C. Johnson.
    Checks "C" programs for syntax errors, type violations, portability problems, and a variety of probable errors.

56. A Tour Through the UNIX "C" Compiler. D.M. Ritchie.
    How the UNIX® operating system "C" compiler works inside.

57. A Tour Through the Portable "C" Compiler. S.C. Johnson.
    How the portable "C" compiler works inside.

58. A Portable Fortran 77 Compiler. S.I. Feldman and P.J. Weinberger.
    The first Fortran 77 compiler, and still one of the best. This version reflects the ongoing work at U.C. Berkeley.

59. Introduction to the f77 I/O Library. D.L. Wasley.
    A description of the revised input/output library for Fortran 77. This document reflects the work carried out at U.C. Berkeley.

60. Berkeley Pascal User's Manual. W.N. Joy, S.L. Graham, C.B. Haley, M.K. McKusick, and P.B. Kessler.
    An interpretive implementation of the reference language.

61. Berkeley Pascal PX Implementation Notes. W.N. Joy and M.K McKusick.
    Describes the implementation of the *px* interpreter which translates Pascal binary code generated by the Pascal translator *pi.*

62. The Programming Language EFL. S.I. Feldman.
    An introduction to a powerful FORTRAN preprocessor providing access to a language with structures much like "C".

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

Supporting Tools

64. YACC — Yet Another Compiler—Compiler. S.C. Johnson.
    Converts a BNF specification of a language and semantic actions written in "C" into a compiler of the language.

65. LEX — A Lexical Analyzer Generator. M.E. Lesk and E. Schmidt.
    Creates a recognizer for a set of regular expressions, each regular expression can be followed by arbitrary "C" code which will be executed when the regular expression is found.

66. RATFOR — A Preprocessor for a Rational Fortran. B.W. Kernighan.
    Converts a Fortran with "C"—like control structures and cosmetics into real, ugly Fortran.

67. The M4 Macro Processor. B.W. Kernighan and D.M. Ritchie.
    M4 is a macro processor useful as a front end for "C", Ratfor, Cobol, and in it own right.

68. SED — A Non-Interactive Text Editor. L.E. McMahon.
    A variant of the editor for processing large inputs.

69. AWK — A Pattern Scanning and Processing Language. A.V. Aho, B.W. Kernighan, and P.J. Weinberger.
    Makes it easy to specify many data transformation and selection operations.

70. DC — An Interactive Desk Calculator. R.H. Morris and L.L. Cherry.
    A super HP calculator, if you don't need floating point.

71. BC — An Arbitrary Precision Desk Calculator Language. L.L. Cherry and R.H. Morris.
    A front end for DC that provides infix notation, control flow, and built-in functions.

72. PROC286 Software Support and the Dosc Command under ICON/UXB. M. Muhlestein.
    Describes the implementation of *dosc* and related software for the ICON computer systems. Intended to assist developers and system administrators in understanding the interface between ICON's UNIX and MS-DOS operating system environment.

73. JOVE Manual for UNIX Users. J. Payne. (4.2BSD Revision by D. Kingston and M. Seiden.
    JOVE is an advanced, self-documenting, customizable real-time display editor. This manual, and the tutorial introduction, is based on the original EMACS editor and user manuals written at M.I.T. by Richard Stallman.

74. An Introduction to the Revision Control System — Revised. W.F. Tichy.
    Describes the benefits of using a source code control system to manage software libraries. Includes a tutorial introduction to the use of RCS.

75. Technical Note on ICON/UXB Magnetic Tape Support. M. Muhlestein.
    Describes certain special features of ICON/UXB (Icon's version of UNIX 4.2BSD) support for magnetic tape.

4.2BSD System Manual

Revised July, 1983

William Joy, Eric Cooper, Robert Fabry,
Samuel Leffler, Kirk McKusick and David Mosher

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

(415) 642-7780

ABSTRACT

This document summarizes the facilities provided by the 4.2BSD version of the UNIX operating system. It does not attempt to act as a tutorial for use of the system nor does it attempt to explain or justify the design of the system facilities. It gives neither motivation nor implementation details, in favor of brevity.

The first section describes the basic kernel functions provided to a UNIX process: process naming and protection, memory management, software interrupts, object references (descriptors), time and statistics functions, and resource controls. These facilities, as well as facilities for bootstrap, shutdown and process accounting, are provided solely by the kernel.

The second section describes the standard system abstractions for files and file systems, communication, terminal handling, and process control and debugging. These facilities are implemented by the operating system or by network server processes.

---

* UNIX is a trademark of Bell Laboratories.

TABLE OF CONTENTS

0. Notation and types

The notation used to describe system calls is a variant of a C language call, consisting of a prototype call followed by declaration of parameters and results. An additional keyword **result**, not part of the normal C language, is used to indicate which of the declared entities receive results. As an example, consider the *read* call, as described in section 2.1:

```
cc = read(fd, buf, nbytes);
result int cc; int fd; result char *buf; int nbytes;
```

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

Description of all error conditions arising from each system call is not provided here; they appear in the programmer's manual. In particular, when accessed from the C language, many calls return a characteristic –1 value when an error occurs, returning the error code in the global variable *errno*. Other languages may present errors in different ways.

A number of system standard types are defined in the include file <sys/types.h> and used in the specifications here and in many C programs. These include **caddr_t** giving a memory address (typically as a character pointer), **off_t** giving a file offset (typically as a long integer), and a set of unsigned types **u_char**, **u_short**, **u_int** and **u_long**, shorthand names for **unsigned char**, **unsigned short**, etc.

1. Kernel primitives

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

The facilities implemented in the kernel are those which define the *UNIX virtual machine* which each process runs in. Like many real machines, this virtual machine has memory management hardware, an interrupt facility, timers and counters. The UNIX virtual machine also allows access to files and other objects through a set of *descriptors*. Each descriptor resembles a device controller, and supports a set of operations. Like devices on real machines, some of which are internal to the machine and some of which are external, parts of the descriptor machinery are built-in to the operating system, while other parts are often implemented in server processes on other machines. The facilities provided through the descriptor machinery are described in section 2.

1.1. Processes and protection

1.1.1. Host and process identifiers

Each UNIX host has associated with it a 32-bit host id, and a host name of up to 255 characters. These are set (by a privileged user) and returned by the calls:

```
sethostid(hostid)
long hostid;

hostid = gethostid();
result long hostid;

sethostname(name, len)
char *name; int len;

len = gethostname(buf, buflen)
result int len; result char *buf; int buflen;
```

On each host runs a set of *processes*. Each process is largely independent of other processes, having its own protection domain, address space, timers, and an independent set of references to system or user implemented objects.

Each process in a host is named by an integer called the *process id*. This number is in the range 1-30000 and is returned by the *getpid* routine:

```
pid = getpid();
result int pid;
```

On each UNIX host this identifier is guaranteed to be unique; in a multi-host environment, the (hostid, process id) pairs are guaranteed unique.

1.1.2. Process creation and termination

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

```
pid = fork();
result int pid;
```

The *fork* call returns twice, once in the parent process, where *pid* is the process identifier of the child, and once in the child process where *pid* is 0. The parent-child relationship induces a hierarchical structure on the set of processes in the system.

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

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

returning 8 bits of exit status to its parent.

When a child process exits or terminates abnormally, the parent process receives information about any event which caused termination of the child process. A second call provides a non-blocking interface and may also be used to retrieve information about resources consumed by the process during its lifetime.

```
#include <sys/wait.h>

pid = wait(astatus);
result int pid; result union wait *astatus;

pid = wait3(astatus, options, arusage);
result int pid; result union waitstatus *astatus;
int options; result struct rusage *arusage;
```

A process can overlay itself with the memory image of another process, passing the newly created process a set of parameters, using the call:

```
execve(name, argv, envp)
char *name, **argv, **envp;
```

The specified *name* must be a file which is in a format recognized by the system, either a binary executable file or a file which causes the execution of a specified interpreter program to process its contents.

1.1.3. User and group ids

Each process in the system has associated with it two user-id's: a *real user id* and a *effective user id*, both non-negative 16 bit integers. Each process has an *real accounting group id* and an *effective accounting group id* and a set of *access group id's*. The group id's are non-negative 16 bit integers. Each process may be in several different access groups, with the maximum concurrent number of access groups a system compilation parameter, the constant NGROUPS in the file <sys/param.h>, guaranteed to be at least 8.

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

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

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

the real and effective accounting group ids by:

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

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

and the access group id set is returned by a *getgroups* call:

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

The user and group id's are assigned at login time using the *setreuid*, *setregid*, and *setgroups* calls:

```
setreuid(ruid, euid);
int ruid, euid;

setregid(rgid, egid);
int rgid, egid;

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

The *setreuid* call sets both the real and effective user-id's, while the *setregid* call sets both the real and effective accounting group id's. Unless the caller is the super-user, *ruid* must be equal to either the current real or effective user-id, and *rgid* equal to either the current real or effective accounting group id. The *setgroups* call is restricted to the super-user.

1.1.4. Process groups

Each process in the system is also normally associated with a *process group*. The group of processes in a process group is sometimes referred to as a *job* and manipulated by high-level system software (such as the shell). The current process group of a process is returned by the *getpgrp* call:

```
pgrp = getpgrp(pid);
result int pgrp; int pid;
```

When a process is in a specific process group it may receive software interrupts affecting the group, causing the group to suspend or resume execution or to be interrupted or terminated. In particular, a system terminal has a process group and only processes which are in the process group of the terminal may read from the terminal, allowing arbitration of terminals among several different jobs.

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

```
setpgrp(pid, pgrp);
int pid, pgrp;
```

Newly created processes are assigned process id's distinct from all processes and process groups, and the same process group as their parent. A normal (unprivileged) process may set its process group equal to its process id. A privileged process may set the process group of any process to any value.

1.2. Memory management†

1.2.1. Text, data and stack

Each process begins execution with three logical areas of memory called text, data and stack. The text area is read-only and shared, while the data and stack areas are private to the process. Both the data and stack areas may be extended and contracted on program request. The call

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

changes the size of the data area by *incr* bytes and returns the new end of the data area, while

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

changes the size of the stack area. The stack area is also automatically extended as needed. On the VAX the text and data areas are adjacent in the P0 region, while the stack section is in the P1 region, and grows downward.

1.2.2. Mapping pages

The system supports sharing of data between processes by allowing pages to be mapped into memory. These mapped pages may be *shared* with other processes or *private* to the process. Protection and sharing options are defined in <mman.h> as:

```
/* protections are chosen from these bits, or-ed together */
#define   PROT_READ         0x4     /* pages can be read */
#define   PROT_WRITE        0x2     /* pages can be written */
#define   PROT_EXEC         0x1     /* pages can be executed */

/* sharing types; choose either SHARED or PRIVATE */
#define   MAP_SHARED        1       /* share changes */
#define   MAP_PRIVATE       2       /* changes are private */
```

The cpu-dependent size of a page is returned by the *getpagesize* system call:

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

The call:

```
mmap(addr, len, prot, share, fd, pos);
caddr_t addr; int len, prot, share, fd; off_t pos;
```

causes the pages starting at *addr* and continuing for *len* bytes to be mapped from the object represented by descriptor *fd*, at absolute position *pos*. The parameter *share* specifies whether modifications made to this mapped copy of the page, are to be kept *private*, or are to be *shared* with other references. The parameter *prot* specifies the accessibility of the mapped pages. The *addr*, *len*, and *pos* parameters must all be multiples of the pagesize.

A process can move pages within its own memory by using the *mremap* call:

```
mremap(addr, len, prot, share, fromaddr);
caddr_t addr; int len, prot, share; caddr_t fromaddr;
```

This call maps the pages starting at *fromaddr* to the address specified by *addr*.

† This section represents the interface planned for later releases of the system. Of the calls described in this section, only *sbrk* and *getpagesize* are included in 4.2BSD.

A mapping can be removed by the call

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

This causes further references to these pages to refer to private pages initialized to zero.

1.2.3. Page protection control

A process can control the protection of pages using the call

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

This call changes the specified pages to have protection *prot*.

1.2.4. Giving and getting advice

A process that has knowledge of its memory behavior may use the *madvise* call:

```
madvise(addr, len, behav);
caddr_t addr; int len, behav;
```

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

```
#define  MADV_NORMAL      0     /* no further special treatment */
#define  MADV_RANDOM      1     /* expect random page references */
#define  MADV_SEQUENTIAL  2     /* expect sequential references */
#define  MADV_WILLNEED    3     /* will need these pages */
#define  MADV_DONTNEED    4     /* don't need these pages */
```

Finally, a process may obtain information about whether pages are core resident by using the call

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

Here the current core residency of the pages is returned in the character array *vec*, with a value of 1 meaning that the page is in-core.

1.3. Signals

1.3.1. Overview

The system defines a set of *signals* that may be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is blocked from further occurrence, the current process context is saved, and a new one is built. A process may specify the *handler* to which a signal is delivered, or specify that the signal is to be *blocked* or *ignored*. A process may also specify that a *default* action is to be taken when signals occur.

Some signals will cause a process to exit when they are not caught. This may be accompanied by creation of a *core* image file, containing the current memory image of the process for use in post-mortem debugging. A process may choose to have signals delivered on a special stack, so that sophisticated software stack manipulations are possible.

All signals have the same *priority*. If multiple signals are pending simultaneously, the order in which they are delivered to a process is implementation specific. Signal routines execute with the signal that caused their invocation *blocked*, but other signals may yet occur. Mechanisms are provided whereby critical sections of code may protect themselves against the occurrence of specified signals.

1.3.2. Signal types

The signals defined by the system fall into one of five classes: hardware conditions, software conditions, input/output notification, process control, or resource control. The set of signals is defined in the file <signal.h>.

Hardware signals are derived from exceptional conditions which may occur during execution. Such signals include SIGFPE representing floating point and other arithmetic exceptions, SIGILL for illegal instruction execution, SIGSEGV for addresses outside the currently assigned area of memory, and SIGBUS for accesses that violate memory protection constraints. Other, more cpu-specific hardware signals exist, such as those for the various customer-reserved instructions on the VAX (SIGIOT, SIGEMT, and SIGTRAP).

Software signals reflect interrupts generated by user request: SIGINT for the normal interrupt signal; SIGQUIT for the more powerful *quit* signal, that normally causes a core image to be generated; SIGHUP and SIGTERM that cause graceful process termination, either because a user has "hung up", or by user or program request; and SIGKILL, a more powerful termination signal which a process cannot catch or ignore. Other software signals (SIGALRM, SIGVTALRM, SIGPROF) indicate the expiration of interval timers.

A process can request notification via a SIGIO signal when input or output is possible on a descriptor, or when a *non-blocking* operation completes. A process may request to receive a SIGURG signal when an urgent condition arises.

A process may be *stopped* by a signal sent to it or the members of its process group. The SIGSTOP signal is a powerful stop signal, because it cannot be caught. Other stop signals SIGTSTP, SIGTTIN, and SIGTTOU are used when a user request, input request, or output request respectively is the reason the process is being stopped. A SIGCONT signal is sent to a process when it is continued from a stopped state. Processes may receive notification with a SIGCHLD signal when a child process changes state, either by stopping or by terminating.

Exceeding resource limits may cause signals to be generated. SIGXCPU occurs when a process nears its CPU time limit and SIGXFSZ warns that the limit on file size creation has been reached.

1.3.3. Signal handlers

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

```
#include <signal.h>

struct sigvec {
        int         (*sv_handler)();
        int         sv_mask;
        int         sv_onstack;
};

sigvec(signo, sv, osv)
int signo; struct sigvec *sv; result struct sigvec *osv;
```

assigns interrupt handler address *sv_handler* to signal *signo*. Each handler address specifies either an interrupt routine for the signal, that the signal is to be ignored, or that a default action (usually process termination) is to occur if the signal occurs. The constants SIG_IGN and SIG_DEF used as values for *sv_handler* cause ignoring or defaulting of a condition. The *sv_mask* and *sv_onstack* values specify the signal mask to be used when the handler is invoked and whether the handler should operate on the normal run-time stack or a special signal stack (see below). If *osv* is non-zero, the previous signal vector is returned.

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently *blocked* by the process then it will be delivered. The process of signal delivery adds the signal to be delivered and those signals specified in the associated signal handler's *sv_mask* to a set of those *masked* for the process, saves the current process context, and places the process in the context of the signal handling routine. The call is arranged so that if the signal handling routine exits normally the signal mask will be restored and the process will resume execution in the original context. If the process wishes to resume in a different context, then it must arrange to restore the signal mask itself.

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

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

```
(*sv_handler)(signo, code, scp);
int signo; long code; struct sigcontext *scp;
```

The *signo* gives the number of the signal that occurred, and the *code*, a word of information supplied by the hardware. The *scp* parameter is a pointer to a machine-dependent structure containing the information for restoring the context before the signal.

1.3.4. Sending signals

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

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

killpgrp(pgrp, signo)
int pgrp, signo;
```

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

Signals are also sent implicitly from a terminal device to the process group associated with the terminal when certain input characters are typed.

1.3.5. Protecting critical sections

To block a section of code against one or more signals, a *sigblock* call may be used to add a set of signals to the existing mask, returning the old mask:

```
oldmask = sigblock(mask);
result long oldmask; long mask;
```

The old mask can then be restored later with *sigsetmask*,

```
oldmask = sigsetmask(mask);
result long oldmask; long mask;
```

The *sigblock* call can be used to read the current mask by specifying an empty *mask*.

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

```
sigpause(mask);
long mask;
```

1.3.6. Signal stacks

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

```
struct sigstack {
        caddr_t     ss_sp;
        int         ss_onstack;
};

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

to provide the system with a stack based at *ss_sp* for delivery of signals. The value *ss_onstack* indicates whether the process is currently on the signal stack, a notion maintained in software by the system.

When a signal is to be delivered, the system checks whether the process is on a signal stack. If not, then the process is switched to the signal stack for delivery, with the return from the signal arranged to restore the previous stack.

If the process wishes to take a non-local exit from the signal routine, or run code from the signal stack that uses a different stack, a *sigstack* call should be used to reset the signal stack.

1.4. Timers

1.4.1. Real time

The system's notion of the current Greenwich time and the current time zone is set and returned by the call by the calls:

```
#include <sys/time.h>

settimeofday(tvp, tzp);
struct timeval *tp;
struct timezone *tzp;

gettimeofday(tp, tzp);
result struct timeval *tp;
result struct timezone *tzp;
```

where the structures are defined in <sys/time.h> as:

```
struct timeval {
        long    tv_sec;         /* seconds since Jan 1, 1970 */
        long    tv_usec;        /* and microseconds */
};

struct timezone {
        int     tz_minuteswest; /* of Greenwich */
        int     tz_dsttime;     /* type of dst correction to apply */
};
```

Earlier versions of UNIX contained only a 1-second resolution version of this call, which remains as a library routine:

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

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

1.4.2. Interval time

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

```
#define ITIMER_REAL     0       /* real time intervals */
#define ITIMER_VIRTUAL  1       /* virtual time intervals */
#define ITIMER_PROF     2       /* user and system virtual time */
```

The ITIMER_REAL timer decrements in real time. It could be used by a library routine to maintain a wakeup service queue. A SIGALRM signal is delivered when this timer expires.

The ITIMER_VIRTUAL timer decrements in process virtual time. It runs only when the process is executing. A SIGVTALRM signal is delivered when it expires.

The ITIMER_PROF timer decrements both in process virtual time and when the system is running on behalf of the process. It is designed to be used by processes to statistically profile their execution. A SIGPROF signal is delivered when it expires.

A timer value is defined by the *itimerval* structure:

```
struct itimerval {
        struct  timeval it_interval; /* timer interval */
        struct  timeval it_value;    /* current value */
};
```

and a timer is set or read by the call:

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

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

The third argument to *setitimer* specifies an optional structure to receive the previous contents of the interval timer. A timer can be disabled by specifying a timer value of 0.

The system rounds argument timer intervals to be not less than the resolution of its clock. This clock resolution can be determined by loading a very small value into a timer and reading the timer back to see what value resulted.

The *alarm* system call of earlier versions of UNIX is provided as a library routine using the ITIMER_REAL timer. The process profiling facilities of earlier versions of UNIX remain because it is not always possible to guarantee the automatic restart of system calls after receipt of a signal.

```
profil(buf, bufsize, offset, scale);
result char *buf; int bufsize, offset, scale;
```

1.5. Descriptors

1.5.1. The reference table

Each process has access to resources through *descriptors*. Each descriptor is a handle allowing the process to reference objects such as files, devices and communications links.

Rather than allowing processes direct access to descriptors, the system introduces a level of indirection, so that descriptors may be shared between processes. Each process has a *descriptor reference table*, containing pointers to the actual descriptors. The descriptors themselves thus have multiple references, and are reference counted by the system.

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

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

and guaranteed to be at least 20. The entries in the descriptor reference table are referred to by small integers; for example if there are 20 slots they are numbered 0 to 19.

1.5.2. Descriptor properties

Each descriptor has a logical set of properties maintained by the system and defined by its *type*. Each type supports a set of operations; some operations, such as reading and writing, are common to several abstractions, while others are unique. The generic operations applying to many of these types are described in section 2.1. Naming contexts, files and directories are described in section 2.2. Section 2.3 describes communications domains and sockets. Terminals and (structured and unstructured) devices are described in section 2.4.

1.5.3. Managing descriptor references

A duplicate of a descriptor reference may be made by doing

```
new = dup(old);
result int new; int old;
```

returning a copy of descriptor reference *old* indistinguishable from the original. The *new* chosen by the system will be the smallest unused descriptor reference slot. A copy of a descriptor reference may be made in a specific slot by doing

```
dup2(old, new);
int old, new;
```

The *dup2* call causes the system to deallocate the descriptor reference current occupying slot *new*, if any, replacing it with a reference to the same descriptor as old. This deallocation is also performed by:

```
close(old);
int old;
```

1.5.4. Multiplexing requests

The system provides a standard way to do synchronous and asynchronous multiplexing of operations.

Synchronous multiplexing is performed by using the *select* call:

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

The *select* call examines the descriptors specified by the sets *in*, *out* and *except*, replacing the

specified bit masks by the subsets that select for input, output, and exceptional conditions respectively (*nd* indicates the size, in bytes, of the bit masks). If any descriptors meet the following criteria, then the number of such descriptors is returned in *nds* and the bit masks are updated.

- A descriptor selects for input if an input oriented operation such as *read* or *receive* is possible, or if a connection request may be accepted (see section 2.3.1.4).
- A descriptor selects for output if an output oriented operation such as *write* or *send* is possible, or if an operation that was "in progress", such as connection establishment, has completed (see section 2.1.3).
- A descriptor selects for an exceptional condition if a condition that would cause a SIGURG signal to be generated exists (see section 1.3.2).

If none of the specified conditions is true, the operation blocks for at most the amount of time specified by *tvp*, or waits for one of the conditions to arise if *tvp* is given as 0.

Options affecting i/o on a descriptor may be read and set by the call:

```
dopt = fcntl(d, cmd, arg)
result int dopt; int d, cmd, arg;

/* interesting values for cmd */
#define  F_SETFL      3     /* set descriptor options */
#define  F_GETFL      4     /* get descriptor options */
#define  F_SETOWN     5     /* set descriptor owner (pid/pgrp) */
#define  F_GETOWN     6     /* get descriptor owner (pid/pgrp) */
```

The F_SETFL *cmd* may be used to set a descriptor in non-blocking i/o mode and/or enable signalling when i/o is possible. F_SETOWN may be used to specify a process or process group to be signalled when using the latter mode of operation.

Operations on non-blocking descriptors will either complete immediately, note an error EWOULDBLOCK, partially complete an input or output operation returning a partial count, or return an error EINPROGRESS noting that the requested operation is in progress. A descriptor which has signalling enabled will cause the specified process and/or process group be signaled, with a SIGIO for input, output, or in-progress operation complete, or a SIGURG for exceptional conditions.

For example, when writing to a terminal using non-blocking output, the system will accept only as much data as there is buffer space for and return; when making a connection on a *socket*, the operation may return indicating that the connection establishment is "in progress". The *select* facility can be used to determine when further output is possible on the terminal, or when the connection establishment attempt is complete.

1.5.5. Descriptor wrapping.†

A user process may build descriptors of a specified type by *wrapping* a communications channel with a system supplied protocol translator:

```
new = wrap(old, proto)
result int new; int old; struct dprop *proto;
```

Operations on the descriptor *old* are then translated by the system provided protocol translator into requests on the underyling object *old* in a way defined by the protocol. The protocols supported by the kernel may vary from system to system and are described in the programmers manual.

Protocols may be based on communications multiplexing or a rights-passing style of handling multiple requests made on the same object. For instance, a protocol for implementing a file

† The facilities described in this section are not included in 4.2BSD.

abstraction may or may not include locally generated "read-ahead" requests. A protocol that provides for read-ahead may provide higher performance but have a more difficult implementation.

Another example is the terminal driving facilities. Normally a terminal is associated with a communications line and the terminal type and standard terminal access protocol is wrapped around a synchronous communications line and given to the user. If a virtual terminal is required, the terminal driver can be wrapped around a communications link, the other end of which is held by a virtual terminal protocol interpreter.

1.6. Resource controls

1.6.1. Process priorities

The system gives CPU scheduling priority to processes that have not used CPU time recently. This tends to favor interactive processes and processes that execute only for short periods. It is possible to determine the priority currently assigned to a process, process group, or the processes of a specified user, or to alter this priority using the calls:

```
#define  PRIO_PROCESS     0     /* process */
#define  PRIO_PGRP        1     /* process group */
#define  PRIO_USER        2     /* user id */

prio = getpriority(which, who);
result int prio; int which, who;

setpriority(which, who, prio);
int which, who, prio;
```

The value *prio* is in the range –20 to 20. The default priority is 0; lower priorities cause more favorable execution. The *getpriority* call returns the highest priority (lowest numerical value) enjoyed by any of the specified processes. The *setpriority* call sets the priorities of all of the specified processes to the specified value. Only the super-user may lower priorities.

1.6.2. Resource utilization

The resources used by a process are returned by a *getrusage* call, returning information in a structure defined in <sys/resource.h>:

```
#define  RUSAGE_SELF      0     /* usage by this process */
#define  RUSAGE_CHILDREN  -1    /* usage by all children */

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

struct rusage {
        struct  timeval ru_utime;   /* user time used */
        struct  timeval ru_stime;   /* system time used */
        int     ru_maxrss;          /* maximum core resident set size: kbytes */
        int     ru_ixrss;           /* integral shared memory size (kbytes*sec) */
        int     ru_idrss;           /* unshared data " */
        int     ru_isrss;           /* unshared stack " */
        int     ru_minflt;          /* page-reclaims */
        int     ru_majflt;          /* page faults */
        int     ru_nswap;           /* swaps */
        int     ru_inblock;         /* block input operations */
        int     ru_oublock;         /* block output " */
        int     ru_msgsnd;          /* messages sent */
        int     ru_msgrcv;          /* messages received */
        int     ru_nsignals;        /* signals received */
        int     ru_nvcsw;           /* voluntary context switches */
        int     ru_nivcsw;          /* involuntary " */
};
```

The *who* parameter specifies whose resource usage is to be returned. The resources used by the current process, or by all the terminated children of the current process may be requested.

1.6.3. Resource limits

The resources of a process for which limits are controlled by the kernel are defined in <sys/resource.h>, and controlled by the *getrlimit* and *setrlimit* calls:

```
#define   RLIMIT_CPU        0       /* cpu time in milliseconds */
#define   RLIMIT_FSIZE      1       /* maximum file size */
#define   RLIMIT_DATA       2       /* maximum data segment size */
#define   RLIMIT_STACK      3       /* maximum stack segment size */
#define   RLIMIT_CORE       4       /* maximum core file size */
#define   RLIMIT_RSS        5       /* maximum resident set size */

#define   RLIM_NLIMITS      6

#define   RLIM_INFINITY     0x7fffffff

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

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

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

Only the super-user can raise the maximum limits. Other users may only alter *rlim_cur* within the range from 0 to *rlim_max* or (irreversibly) lower *rlim_max*.

1.7. System operation support

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

1.7.1. Bootstrap operations

The call

```
mount(blkdev, dir, ronly);
char *blkdev, *dir; int ronly;
```

extends the UNIX name space. The *mount* call specifies a block device *blkdev* containing a UNIX file system to be made available starting at *dir*. If *ronly* is set then the file system is read-only; writes to the file system will not be permitted and access times will not be updated when files are referenced. *Dir* is normally a name in the root directory.

The call

```
swapon(blkdev, size);
char *blkdev; int size;
```

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

1.7.2. Shutdown operations

The call

```
unmount(dir);
char *dir;
```

unmounts the file system mounted on *dir*. This call will succeed only if the file system is not currently being used.

The call

```
sync();
```

schedules input/output to clean all system buffer caches. (This call does not require priveleged status.)

The call

```
reboot(how)
int how;
```

causes a machine halt or reboot. The call may request a reboot by specifying *how* as RB_AUTOBOOT, or that the machine be halted with RB_HALT. These constants are defined in <sys/reboot.h>.

1.7.3. Accounting

The system optionally keeps an accounting record in a file for each process that exits on the system. The format of this record is beyond the scope of this document. The accounting may be enabled to a file *name* by doing

```
acct(path);
char *path;
```

If *path* is null, then accounting is disabled. Otherwise, the named file becomes the accounting file.

2. System facilities

This section discusses the system facilities that are not considered part of the kernel.

The system abstractions described are:

Directory contexts

A directory context is a position in the UNIX file system name space. Operations on files and other named objects in a file system are always specified relative to such a context.

Files

Files are used to store uninterpreted sequence of bytes on which random access *reads* and *writes* may occur. Pages from files may also be mapped into process address space. A directory may be read as a file†.

Communications domains

A communications domain represents an interprocess communications environment, such as the communications facilities of the UNIX system, communications in the INTERNET, or the resource sharing protocols and access rights of a resource sharing system on a local network.

Sockets

A socket is an endpoint of communication and the focal point for IPC in a communications domain. Sockets may be created in pairs, or given names and used to rendezvous with other sockets in a communications domain, accepting connections from these sockets or exchanging messages with them. These operations model a labeled or unlabeled communications graph, and can be used in a wide variety of communications domains. Sockets can have different *types* to provide different semantics of communication, increasing the flexibility of the model.

Terminals and other devices

Devices include terminals, providing input editing and interrupt generation and output flow control and editing, magnetic tapes, disks and other peripherals. They often support the generic *read* and *write* operations as well as a number of *ioctl*s.

Processes

Process descriptors provide facilities for control and debugging of other processes.

† Support for mapping files is not included in the 4.2 release.

2.1. Generic operations

Many system abstractions support the operations *read*, *write* and *ioctl*. We describe the basics of these common primitives here. Similarly, the mechanisms whereby normally synchronous operations may occur in a non-blocking or asynchronous fashion are common to all system-defined abstractions and are described here.

2.1.1. Read and write

The *read* and *write* system calls can be applied to communications channels, files, terminals and devices. They have the form:

```
cc = read(fd, buf, nbytes);
result int cc; int fd; result caddr_t buf; int nbytes;

cc = write(fd, buf, nbytes);
result int cc; int fd; caddr_t buf; int nbytes;
```

The *read* call transfers as much data as possible from the object defined by *fd* to the buffer at address *buf* of size *nbytes*. The number of bytes transferred is returned in *cc*, which is –1 if a return occurred before any data was transferred because of an error or use of non-blocking operations.

The *write* call transfers data from the buffer to the object defined by *fd*. Depending on the type of *fd*, it is possible that the *write* call will accept some portion of the provided bytes; the user should resubmit the other bytes in a later request in this case. Error returns because of interrupted or otherwise incomplete operations are possible.

Scattering of data on input or gathering of data for output is also possible using an array of input/output vector descriptors. The type for the descriptors is defined in <sys/uio.h> as:

```
struct iovec {
        caddr_t     iov_msg;        /* base of a component */
        int         iov_len;        /* length of a component */
};
```

The calls using an array of descriptors are:

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

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

Here *iovlen* is the count of elements in the *iov* array.

2.1.2. Input/output control

Control operations on an object are performed by the *ioctl* operation:

```
ioctl(fd, request, buffer);
int fd, request; caddr_t buffer;
```

This operation causes the specified *request* to be performed on the object *fd*. The *request* parameter specifies whether the argument buffer is to be read, written, read and written, or is not needed, and also the size of the buffer, as well as the request. Different descriptor types and subtypes within descriptor types may use distinct *ioctl* requests. For example, operations on terminals control flushing of input and output queues and setting of terminal parameters; operations on disks cause formatting operations to occur; operations on tapes control tape positioning.

The names for basic control operations are defined in <sys/ioctl.h>.

2.1.3. Non-blocking and asynchronous operations

A process that wishes to do non-blocking operations on one of its descriptors sets the descriptor in non-blocking mode as described in section 1.5.4. Thereafter the *read* call will return a specific EWOULDBLOCK error indication if there is no data to be *read*. The process may *dselect* the associated descriptor to determine when a read is possible.

Output attempted when a descriptor can accept less than is requested will either accept some of the provided data, returning a shorter than normal length, or return an error indicating that the operation would block. More output can be performed as soon as a *select* call indicates the object is writeable.

Operations other than data input or output may be performed on a descriptor in a non-blocking fashion. These operations will return with a characteristic error indicating that they are in progress if they cannot return immediately. The descriptor may then be *selected* for *write* to find out when the operation can be retried. When *select* indicates the descriptor is writeable, a respecification of the original operation will return the result of the operation.

2.2. File system

2.2.1. Overview

The file system abstraction provides access to a hierarchical file system structure. The file system contains directories (each of which may contain other sub-directories) as well as files and references to other objects such as devices and inter-process communications sockets.

Each file is organized as a linear array of bytes. No record boundaries or system related information is present in a file. Files may be read and written in a random-access fashion. The user may read the data in a directory as though it were an ordinary file to determine the names of the contained files, but only the system may write into the directories. The file system stores only a small amount of ownership, protection and usage information with a file.

2.2.2. Naming

The file system calls take *path name* arguments. These consist of a zero or more component *file names* separated by "/" characters, where each file name is up to 255 ASCII characters excluding null and "/".

Each process always has two naming contexts: one for the root directory of the file system and one for the current working directory. These are used by the system in the filename translation process. If a path name begins with a "/", it is called a full path name and interpreted relative to the root directory context. If the path name does not begin with a "/" it is called a relative path name and interpreted relative to the current directory context.

The system limits the total length of a path name to 1024 characters.

The file name ".." in each directory refers to the parent directory of that directory. The parent directory of a file system is always the systems root directory.

The calls

```
chdir(path);
char *path;

chroot(path)
char *path;
```

change the current working directory and root directory context of a process. Only the super-user can change the root directory context of a process.

2.2.3. Creation and removal

The file system allows directories, files, special devices, and "portals" to be created and removed from the file system.

2.2.3.1. Directory creation and removal

A directory is created with the *mkdir* system call:

```
mkdir(path, mode);
char *path; int mode;
```

and removed with the *rmdir* system call:

```
rmdir(path);
char *path;
```

A directory must be empty if it is to be deleted.

2.2.3.2. File creation

Files are created with the *open* system call,

```
fd = open(path, oflag, mode);
result int fd; char *path; int oflag, mode;
```

The *path* parameter specifies the name of the file to be created. The *oflag* parameter must include O_CREAT from below to cause the file to be created. The protection for the new file is specified in *mode*. Bits for *oflag* are defined in <sys/file.h>:

```
#define  O_RDONLY    000    /* open for reading */
#define  O_WRONLY    001    /* open for writing */
#define  O_RDWR      002    /* open for read & write */
#define  O_NDELAY    004    /* non-blocking open */
#define  O_APPEND    010    /* append on each write */
#define  O_CREAT     01000  /* open with file create */
#define  O_TRUNC     02000  /* open with truncation */
#define  O_EXCL      04000  /* error on create if file exists */
```

One of O_RDONLY, O_WRONLY and O_RDWR should be specified, indicating what types of operations are desired to be performed on the open file. The operations will be checked against the user's access rights to the file before allowing the *open* to succeed. Specifying O_APPEND causes writes to automatically append to the file. The flag O_CREAT causes the file to be created if it does not exist, with the specified *mode*, owned by the current user and the group of the containing directory.

If the open specifies to create the file with O_EXCL and the file already exists, then the *open* will fail without affecting the file in any way. This provides a simple exclusive access facility.

2.2.3.3. Creating references to devices

The file system allows entries which reference peripheral devices. Peripherals are distinguished as *block* or *character* devices according by their ability to support block-oriented operations. Devices are identified by their "major" and "minor" device numbers. The major device number determines the kind of peripheral it is, while the minor device number indicates one of possibly many peripherals of that kind. Structured devices have all operations performed internally in "block" quantities while unstructured devices often have a number of special *ioctl* operations, and may have input and output performed in large units. The *mknod* call creates special entries:

```
mknod(path, mode, dev);
char *path; int mode, dev;
```

where *mode* is formed from the object type and access permissions. The parameter *dev* is a configuration dependent parameter used to identify specific character or block i/o devices.

2.2.3.4. Portal creation†

The call

```
fd = portal(name, server, param, dtype, protocol, domain, socktype)
result int fd; char *name, *server, *param; int dtype, protocol;
int domain, socktype;
```

places a *name* in the file system name space that causes connection to a server process when the name is used. The portal call returns an active portal in *fd* as though an access had occurred to activate an inactive portal, as now described.

† The *portal* call is not implemented in 4.2BSD.

When an inactive portal is accesseed, the system sets up a socket of the specified *socktype* in the specified communications *domain* (see section 2.3), and creates the *server* process, giving it the specified *param* as argument to help it identify the portal, and also giving it the newly created socket as descriptor number 0. The accessor of the portal will create a socket in the same *domain* and *connect* to the server. The user will then *wrap* the socket in the specified *protocol* to create an object of the required descriptor type *dtype* and proceed with the operation which was in progress before the portal was encountered.

While the server process holds the socket (which it received as *fd* from the *portal* call on descriptor 0 at activation) further references will result in connections being made to the same socket.

2.2.3.5. File, device, and portal removal

A reference to a file, special device or portal may be removed with the *unlink* call,

```
unlink(path);
char *path;
```

The caller must have write access to the directory in which the file is located for this call to be successful.

2.2.4. Reading and modifying file attributes

Detailed information about the attributes of a file may be obtained with the calls:

```
#include <sys/stat.h>

stat(path, stb);
char *path; result struct stat *stb;

fstat(fd, stb);
int fd; result struct stat *stb;
```

The *stat* structure includes the file type, protection, ownership, access times, size, and a count of hard links. If the file is a symbolic link, then the status of the link itself (rather than the file the link references) may be found using the *lstat* call:

```
lstat(path, stb);
char *path; result struct stat *stb;
```

Newly created files are assigned the user id of the process that created it and the group id of the directory in which it was created. The ownership of a file may be changed by either of the calls

```
chown(path, owner, group);
char *path; int owner, group;

fchown(fd, owner, group);
int fd, owner, group;
```

In addition to ownership, each file has three levels of access protection associated with it. These levels are owner relative, group relative, and global (all users and groups). Each level of access has separate indicators for read permission, write permission, and execute permission. The protection bits associated with a file may be set by either of the calls:

```
chmod(path, mode);
char *path; int mode;

fchmod(fd, mode);
int fd, mode;
```

where *mode* is a value indicating the new protection of the file. The file mode is a three digit octal number. Each digit encodes read access as 4, write access as 2 and execute access as 1, or'ed together. The 0700 bits describe owner access, the 070 bits describe the access rights for processes in the same group as the file, and the 07 bits describe the access rights for other processes.

Finally, the access and modify times on a file may be set by the call:

```
utimes(path, tvp)
char *path; struct timeval *tvp[2];
```

This is particularly useful when moving files between media, to preserve relationships between the times the file was modified.

2.2.5. Links and renaming

Links allow multiple names for a file to exist. Links exist independently of the file linked to.

Two types of links exist, *hard* links and *symbolic* links. A hard link is a reference counting mechanism that allows a file to have multiple names within the same file system. Symbolic links cause string substitution during the pathname interpretation process.

Hard links and symbolic links have different properties. A hard link insures the target file will always be accessible, even after its original directory entry is removed; no such guarantee exists for a symbolic link. Symbolic links can span file systems boundaries.

The following calls create a new link, named *path2*, to *path1*:

```
link(path1, path2);
char *path1, *path2;

symlink(path1, path2);
char *path1, *path2;
```

The *unlink* primitive may be used to remove either type of link.

If a file is a symbolic link, the "value" of the link may be read with the *readlink* call,

```
len = readlink(path, buf, bufsize);
result int len; result char *path, *buf; int bufsize;
```

This call returns, in *buf*, the null-terminated string substituted into pathnames passing through *path*.

Atomic renaming of file system resident objects is possible with the *rename* call:

```
rename(oldname, newname);
char *oldname, *newname;
```

where both *oldname* and *newname* must be in the same file system. If *newname* exists and is a directory, then it must be empty.

2.2.6. Extension and truncation

Files are created with zero length and may be extended simply by writing or appending to them. While a file is open the system maintains a pointer into the file indicating the current location in the file associated with the descriptor. This pointer may be moved about in the file in a random access fashion. To set the current offset into a file, the *lseek* call may be used,

```
oldoffset = lseek(fd, offset, type);
result off_t oldoffset; int fd; off_t offset; int type;
```

where *type* is given in <sys/file.h> as one of,

```
#define  L_SET      0    /* set absolute file offset */
#define  L_INCR     1    /* set file offset relative to current position */
#define  L_XTND     2    /* set offset relative to end-of-file */
```

The call "lseek(fd, 0, L_INCR)" returns the current offset into the file.

Files may have "holes" in them. Holes are void areas in the linear extent of the file where data has never been written. These may be created by seeking to a location in a file past the current end-of-file and writing. Holes are treated by the system as zero valued bytes.

A file may be truncated with either of the calls:

```
truncate(path, length);
char *path; int length;

ftruncate(fd, length);
int fd, length;
```

reducing the size of the specified file to *length* bytes.

2.2.7. Checking accessibility

A process running with different real and effective user ids may interrogate the accessibility of a file to the real user by using the *access* call:

```
accessible = access(path, how);
result int accessible; char *path; int how;
```

Here *how* is constructed by or'ing the following bits, defined in <sys/file.h>:

```
#define  F_OK       0    /* file exists */
#define  X_OK       1    /* file is executable */
#define  W_OK       2    /* file is writable */
#define  R_OK       4    /* file is readable */
```

The presence or absence of advisory locks does not affect the result of *access*.

2.2.8. Locking

The file system provides basic facilities that allow cooperating processes to synchronize their access to shared files. A process may place an advisory *read* or *write* lock on a file, so that other cooperating processes may avoid interfering with the process' access. This simple mechanism provides locking with file granularity. More granular locking can be built using the IPC facilities to provide a lock manager. The system does not force processes to obey the locks; they are of an advisory nature only.

Locking is performed after an *open* call by applying the *flock* primitive,

```
flock(fd, how);
int fd, how;
```

where the *how* parameter is formed from bits defined in <sys/file.h>:

```
#define  LOCK_SH    1    /* shared lock */
#define  LOCK_EX    2    /* exclusive lock */
#define  LOCK_NB    4    /* don't block when locking */
#define  LOCK_UN    8    /* unlock */
```

Successive lock calls may be used to increase or decrease the level of locking. If an object is

currently locked by another process when a *flock* call is made, the caller will be blocked until the current lock owner releases the lock; this may be avoided by including LOCK_NB in the *how* parameter. Specifying LOCK_UN removes all locks associated with the descriptor. Advisory locks held by a process are automatically deleted when the process terminates.

2.2.9. Disk quotas

As an optional facility, each file system may be requested to impose limits on a user's disk usage. Two quantities are limited: the total amount of disk space which a user may allocate in a file system and the total number of files a user may create in a file system. Quotas are expressed as *hard* limits and *soft* limits. A hard limit is always imposed; if a user would exceed a hard limit, the operation which caused the resource request will fail. A soft limit results in the user receiving a warning message, but with allocation succeeding. Facilities are provided to turn soft limits into hard limits if a user has exceeded a soft limit for an unreasonable period of time.

To enable disk quotas on a file system the *setquota* call is used:

```
setquota(special, file)
char *special, *file;
```

where *special* refers to a structured device file where a mounted file system exists, and *file* refers to a disk quota file (residing on the file system associated with *special*) from which user quotas should be obtained. The format of the disk quota file is implementation dependent.

To manipulate disk quotas the *quota* call is provided:

```
#include <sys/quota.h>

quota(cmd, uid, arg, addr)
int cmd, uid, arg; caddr_t addr;
```

The indicated *cmd* is applied to the user ID *uid*. The parameters *arg* and *addr* are command specific. The file <sys/quota.h> contains definitions pertinent to the use of this call.

2.3. Interprocess communications

2.3.1. Interprocess communication primitives

2.3.1.1. Communication domains

The system provides access to an extensible set of communication *domains*. A communication domain is identified by a manifest constant defined in the file <sys/socket.h>. Important standard domains supported by the system are the "unix" domain, AF_UNIX, for communication within the system, and the "internet" domain for communication in the DARPA internet, AF_INET. Other domains can be added to the system.

2.3.1.2. Socket types and protocols

Within a domain, communication takes place between communication endpoints known as *sockets*. Each socket has the potential to exchange information with other sockets within the domain.

Each socket has an associated abstract type, which describes the semantics of communication using that socket. Properties such as reliability, ordering, and prevention of duplication of messages are determined by the type. The basic set of socket types is defined in <sys/socket.h>:

```
/* Standard socket types */
#define  SOCK_DGRAM      1     /* datagram */
#define  SOCK_STREAM     2     /* virtual circuit */
#define  SOCK_RAW        3     /* raw socket */
#define  SOCK_RDM        4     /* reliably-delivered message */
#define  SOCK_SEQPACKET  5     /* sequenced packets */
```

The SOCK_DGRAM type models the semantics of datagrams in network communication: messages may be lost or duplicated and may arrive out-of-order. The SOCK_RDM type models the semantics of reliable datagrams: messages arrive unduplicated and in-order, the sender is notified if messages are lost. The *send* and *receive* operations (described below) generate reliable/unreliable datagrams. The SOCK_STREAM type models connection-based virtual circuits: two-way byte streams with no record boundaries. The SOCK_SEQPACKET type models a connection-based, full-duplex, reliable, sequenced packet exchange; the sender is notified if messages are lost, and messages are never duplicated or presented out-of-order. Users of the last two abstractions may use the facilities for out-of-band transmission to send out-of-band data.

SOCK_RAW is used for unprocessed access to internal network layers and interfaces; it has no specific semantics.

Other socket types can be defined.†

Each socket may have a concrete *protocol* associated with it. This protocol is used within the domain to provide the semantics required by the socket type. For example, within the "internet" domain, the SOCK_DGRAM type may be implemented by the UDP user datagram protocol, and the SOCK_STREAM type may be implemented by the TCP transmission control protocol, while no standard protocols to provide SOCK_RDM or SOCK_SEQPACKET sockets exist.

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;
```

† 4.2BSD does not support the SOCK_RDM and SOCK_SEQPACKET types.

An unconnected socket descriptor may yield a connected socket descriptor in one of two ways: either by actively connecting to another socket, or by becoming associated with a name in the communications domain and *accepting* a connection from another socket.

To accept connections, a socket must first have a binding to a name within the communications domain. Such a binding is established by a *bind* call:

```
bind(s, name, namelen);
int s; char *name; int namelen;
```

A socket's bound name may be retrieved with a *getsockname* call:

```
getsockname(s, name, namelen);
int s; result caddr_t name; result int *namelen;
```

while the peer's name can be retrieved with *getpeername*:

```
getpeername(s, name, namelen);
int s; result caddr_t name; result int *namelen;
```

Domains may support sockets with several names.

2.3.1.4. Accepting connections

Once a binding is made, it is possible to *listen* for connections:

```
listen(s, backlog);
int s, backlog;
```

The *backlog* specifies the maximum count of connections that can be simultaneously queued awaiting acceptance.

An *accept* call:

```
t = accept(s, name, anamelen);
result int t; int s; result caddr_t name; result int *anamelen;
```

returns a descriptor for a new, connected, socket from the queue of pending connections on *s*.

2.3.1.5. Making connections

An active connection to a named socket is made by the *connect* call:

```
connect(s, name, namelen);
int s; caddr_t name; int namelen;
```

It is also possible to create connected pairs of sockets without using the domain's name space to rendezvous; this is done with the *socketpair* call†:

```
socketpair(d, type, protocol, sv);
int d, type, protocol; result int sv[2];
```

Here the returned *sv* descriptors correspond to those obtained with *accept* and *connect*.

The call

```
pipe(pv)
result int pv[2];
```

creates a pair of SOCK_STREAM sockets in the UNIX domain, with pv[0] only writeable and pv[1] only readable.

† 4.2BSD supports *socketpair* creation only in the "unix" communication domain.

2.3.1.6. Sending and receiving data

Messages may be sent from a socket by:

```
cc = sendto(s, buf, len, flags, to, tolen);
result int cc; int s; caddr_t buf; int len, flags; caddr_t to; int tolen;
```

if the socket is not connected or:

```
cc = send(s, buf, len, flags);
result int cc; int s; caddr_t buf; int len, flags;
```

if the socket is connected. The corresponding receive primitives are:

```
msglen = recvfrom(s, buf, len, flags, from, fromlenaddr);
result int msglen; int s; result caddr_t buf; int len, flags;
result caddr_t from; result int *fromlenaddr;
```

and

```
msglen = recv(s, buf, len, flags);
result int msglen; int s; result caddr_t buf; int len, flags;
```

In the unconnected case, the parameters *to* and *tolen* specify the destination or source of the message, while the *from* parameter stores the source of the message, and **fromlenaddr* initially gives the size of the *from* buffer and is updated to reflect the true length of the *from* address.

All calls cause the message to be received in or sent from the message buffer of length *len* bytes, starting at address *buf*. The *flags* specify peeking at a message without reading it or sending or receiving high-priority out-of-band messages, as follows:

```
#define MSG_PEEK        0x1     /* peek at incoming message */
#define MSG_OOB         0x2     /* process out-of-band data */
```

2.3.1.7. Scatter/gather and exchanging access rights

It is possible scatter and gather data and to exchange access rights with messages. When either of these operations is involved, the number of parameters to the call becomes large. Thus the system defines a message header structure, in <sys/socket.h>, which can be used to conveniently contain the parameters to the calls:

```
struct msghdr {
        caddr_t     msg_name;           /* optional address */
        int         msg_namelen;        /* size of address */
        struct      iov *msg_iov;       /* scatter/gather array */
        int         msg_iovlen;         /* # elements in msg_iov */
        caddr_t     msg_accrights;      /* access rights sent/received */
        int         msg_accrightslen;   /* size of msg_accrights */
};
```

Here *msg_name* and *msg_namelen* specify the source or destination address if the socket is unconnected; *msg_name* may be given as a null pointer if no names are desired or required. The *msg_iov* and *msg_iovlen* describe the scatter/gather locations, as described in section 2.1.3. Access rights to be sent along with the message are specified in *msg_accrights*, which has length *msg_accrightslen*. In the "unix" domain these are an array of integer descriptors, taken from the sending process and duplicated in the receiver.

This structure is used in the operations *sendmsg* and *recvmsg*:

```
sendmsg(s, msg, flags);
int s; struct msghdr *msg; int flags;

msglen = recvmsg(s, msg, flags);
result int msglen; int s; result struct msghdr *msg; int flags;
```

2.3.1.8. Using read and write with sockets

The normal UNIX *read* and *write* calls may be applied to connected sockets and translated into *send* and *receive* calls from or to a single area of memory and discarding any rights received. A process may operate on a virtual circuit socket, a terminal or a file with blocking or non-blocking input/output operations without distinguishing the descriptor type.

2.3.1.9. Shutting down halves of full-duplex connections

A process that has a full-duplex socket such as a virtual circuit and no longer wishes to read from or write to this socket can give the call:

```
shutdown(s, direction);
int s, direction;
```

where *direction* is 0 to not read further, 1 to not write further, or 2 to completely shut the connection down.

2.3.1.10. Socket and protocol options

Sockets, and their underlying communication protocols, may support *options*. These options may be used to manipulate implementation specific or non-standard facilities. The *getsockopt* and *setsockopt* calls are used to control options:

```
getsockopt(s, level, optname, optval, optlen)
int s, level, optname; result caddr_t optval; result int *optlen;

setsockopt(s, level, optname, optval, optlen)
int s, level, optname; caddr_t optval; int optlen;
```

The option *optname* is interpreted at the indicated protocol *level* for socket *s*. If a value is specified with *optval* and *optlen*, it is interpreted by the software operating at the specified *level*. The *level* SOL_SOCKET is reserved to indicate options maintained by the socket facilities. Other *level* values indicate a particular protocol which is to act on the option request; these values are normally interpreted as a "protocol number".

2.3.2. UNIX domain

This section describes briefly the properties of the UNIX communications domain.

2.3.2.1. Types of sockets

In the UNIX domain, the SOCK_STREAM abstraction provides pipe-like facilities, while SOCK_DGRAM provides (usually) reliable message-style communications.

2.3.2.2. Naming

Socket names are strings and may appear in the UNIX file system name space through portals†.

† The 4.2BSD implementation of the UNIX domain embeds bound sockets in the UNIX file system name space; this is a side effect of the implementation.

2.3.2.3. Access rights transmission

The ability to pass UNIX descriptors with messages in this domain allows migration of service within the system and allows user processes to be used in building system facilities.

2.3.3. INTERNET domain

This section describes briefly how the INTERNET domain is mapped to the model described in this section. More information will be found in the document describing the network implementation in 4.2BSD.

2.3.3.1. Socket types and protocols

SOCK_STREAM is supported by the INTERNET TCP protocol; SOCK_DGRAM by the UDP protocol. The SOCK_SEQPACKET has no direct INTERNET family analogue; a protocol based on one from the XEROX NS family and layered on top of IP could be implemented to fill this gap.

2.3.3.2. Socket naming

Sockets in the INTERNET domain have names composed of the 32 bit internet address, and a 16 bit port number. Options may be used to provide source routing for the address, security options, or additional address for subnets of INTERNET for which the basic 32 bit addresses are insufficient.

2.3.3.3. Access rights transmission

No access rights transmission facilities are provided in the INTERNET domain.

2.3.3.4. Raw access

The INTERNET domain allows the super-user access to the raw facilities of the various network interfaces and the various internal layers of the protocol implementation. This allows administrative and debugging functions to occur. These interfaces are modeled as SOCK_RAW sockets.

2.4. Terminals and Devices

2.4.1. Terminals

Terminals support *read* and *write* i/o operations, as well as a collection of terminal specific *ioctl* operations, to control input character editing, and output delays.

2.4.1.1. Terminal input

Terminals are handled according to the underlying communication characteristics such as baud rate and required delays, and a set of software parameters.

2.4.1.1.1. Input modes

A terminal is in one of three possible modes: *raw*, *cbreak*, or *cooked*. In raw mode all input is passed through to the reading process immediately and without interpretation. In cbreak mode, the handler interprets input only by looking for characters that cause interrupts or output flow control; all other characters are made available as in raw mode. In cooked mode, input is processed to provide standard line-oriented local editing functions, and input is presented on a line-by-line basis.

2.4.1.1.2. Interrupt characters

Interrupt characters are interpreted by the terminal handler only in cbreak and cooked modes, and cause a software interrupt to be sent to all processes in the process group associated with the terminal. Interrupt characters exist to send SIGINT and SIGQUIT signals, and to stop a process group with the SIGTSTP signal either immediately, or when all input up to the stop character has been read.

2.4.1.1.3. Line editing

When the terminal is in cooked mode, editing of an input line is performed. Editing facilities allow deletion of the previous character or word, or deletion of the current input line. In addition, a special character may be used to reprint the current input line after some number of editing operations have been applied.

Certain other characters are interpreted specially when a process is in cooked mode. The *end of line* character determines the end of an input record. The *end of file* character simulates an end of file occurrence on terminal input. Flow control is provided by *stop output* and *start output* control characters. Output may be flushed with the *flush output* character; and a *literal character* may be used to force literal input of the immediately following character in the input line.

2.4.1.2. Terminal output

On output, the terminal handler provides some simple formatting services. These include converting the carriage return character to the two character return-linefeed sequence, displaying non-graphic ASCII characters as "^character", inserting delays after certain standard control characters, expanding tabs, and providing translations for upper-case only terminals.

2.4.1.3. Terminal control operations

When a terminal is first opened it is initialized to a standard state and configured with a set of standard control, editing, and interrupt characters. A process may alter this configuration with certain control operations, specifying parameters in a standard structure:

```
struct ttymode {
        short       tt_ispeed;      /* input speed */
        int         tt_iflags;      /* input flags */
        short       tt_ospeed;      /* output speed */
        int         tt_oflags;      /* output flags */
};
```

and "special characters" are specified with the *ttychars* structure,

```
struct ttychars {
        char        tc_erasec;      /* erase char */
        char        tc_killc;       /* erase line */
        char        tc_intrc;       /* interrupt */
        char        tc_quitc;       /* quit */
        char        tc_startc;      /* start output */
        char        tc_stopc;       /* stop output */
        char        tc_eofc;        /* end-of-file */
        char        tc_brkc;        /* input delimiter (like nl) */
        char        tc_suspc;       /* stop process signal */
        char        tc_dsuspc;      /* delayed stop process signal */
        char        tc_rprntc;      /* reprint line */
        char        tc_flushc;      /* flush output (toggles) */
        char        tc_werasc;      /* word erase */
        char        tc_lnextc;      /* literal next character */
};
```

2.4.1.4. Terminal hardware support

The terminal handler allows a user to access basic hardware related functions; e.g. line speed, modem control, parity, and stop bits. A special signal, SIGHUP, is automatically sent to processes in a terminal's process group when a carrier transition is detected. This is normally associated with a user hanging up on a modem controlled terminal line.

2.4.2. Structured devices

Structures devices are typified by disks and magnetic tapes, but may represent any random-access device. The system performs read-modify-write type buffering actions on block devices to allow them to be read and written in a totally random access fashion like ordinary files. File systems are normally created in block devices.

2.4.3. Unstructured devices

Unstructured devices are those devices which do not support block structure. Familiar unstructured devices are raw communications lines (with no terminal handler), raster plotters, magnetic tape and disks unfettered by buffering and permitting large block input/output and positioning and formatting commands.

2.5. Process and kernel descriptors

The status of the facilities in this section is still under discussion. The *ptrace* facility of 4.1BSD is provided in 4.2BSD. Planned enhancements would allow a descriptor based process control facility.

I. Summary of facilities

1. Kernel primitives

1.1. Process naming and protection

sethostid	set UNIX host id
gethostid	get UNIX host id
sethostname	set UNIX host name
gethostname	get UNIX host name
getpid	get process id
fork	create new process
exit	terminate a process
execve	execute a different process
getuid	get user id
geteuid	get effective user id
setreuid	set real and effective user id's
getgid	get accounting group id
getegid	get effective accounting group id
getgroups	get access group set
setregid	set real and effective group id's
setgroups	set access group set
getpgrp	get process group
setpgrp	set process group

1.2 Memory management

<mman.h>	memory management definitions
sbrk	change data section size
sstk†	change stack section size
getpagesize	get memory page size
mmap†	map pages of memory
mremap†	remap pages in memory
munmap†	unmap memory
mprotect†	change protection of pages
madvise†	give memory management advice
mincore†	determine core residency of pages

1.3 Signals

<signal.h>	signal definitions
sigvec	set handler for signal
kill	send signal to process
killpgrp	send signal to process group
sigblock	block set of signals
sigsetmask	restore set of blocked signals
sigpause	wait for signals
sigstack	set software stack for signals

1.4 Timing and statistics

<sys/time.h>	time-related definitions
gettimeofday	get current time and timezone
settimeofday	set current time and timezone
getitimer	read an interval timer
setitimer	get and set an interval timer

† Not supported in 4.2BSD.

profil	profile process

1.5 Descriptors

getdtablesize	descriptor reference table size
dup	duplicate descriptor
dup2	duplicate to specified index
close	close descriptor
select	multiplex input/output
fcntl	control descriptor options
wrap†	wrap descriptor with protocol

1.6 Resource controls

<sys/resource.h>	resource-related definitions
getpriority	get process priority
setpriority	set process priority
getrusage	get resource usage
getrlimit	get resource limitations
setrlimit	set resource limitations

1.7 System operation support

mount	mount a device file system
swapon	add a swap device
umount	umount a file system
sync	flush system caches
reboot	reboot a machine
acct	specify accounting file

2. System facilities

2.1 Generic operations

read	read data
write	write data
<sys/uio.h>	scatter-gather related definitions
readv	scattered data input
writev	gathered data output
<sys/ioctl.h>	standard control operations
ioctl	device control operation

2.2 File system

Operations marked with a * exist in two forms: as shown, operating on a file name, and operating on a file descriptor, when the name is preceded with a "f".

<sys/file.h>	file system definitions
chdir	change directory
chroot	change root directory
mkdir	make a directory
rmdir	remove a directory
open	open a new or existing file
mknod	make a special file
portal†	make a portal entry
unlink	remove a link
stat*	return status for a file

† Not supported in 4.2BSD.

lstat	returned status of link
chown*	change owner
chmod*	change mode
utimes	change access/modify times
link	make a hard link
symlink	make a symbolic link
readlink	read contents of symbolic link
rename	change name of file
lseek	reposition within file
truncate*	truncate file
access	determine accessibility
flock	lock a file

2.3 Communications

<sys/socket.h>	standard definitions
socket	create socket
bind	bind socket to name
getsockname	get socket name
listen	allow queueing of connections
accept	accept a connection
connect	connect to peer socket
socketpair	create pair of connected sockets
sendto	send data to named socket
send	send data to connected socket
recvfrom	receive data on unconnected socket
recv	receive data on connected socket
sendmsg	send gathered data and/or rights
recvmsg	receive scattered data and/or rights
shutdown	partially close full-duplex connection
getsockopt	get socket option
setsockopt	set socket option

2.5 Terminals, block and character devices

2.4 Processes and kernel hooks

Fsck – The UNIX† File System Check Program

Revised December 6, 1985

Ronald Holt Jr.

Icon Systems and Software
Software Development Group
Orem, Utah

ABSTRACT

This document reflects the use of *fsck* with the Icon 4.2BSD file system organization. This is a revision of the original paper written by T. J. Kowalski.

File System Check Program (*fsck*) is an interactive file system check and repair program. *Fsck* uses the redundant structural information in the UNIX file system to perform several consistency checks. If an inconsistency is detected, it is reported to the operator, who may elect to fix or ignore each inconsistency. These inconsistencies result from the permanent interruption of the file system updates, which are performed every time a file is modified. Unless there has been a hardware failure, *fsck* is able to repair corrupted file systems using procedures based upon the order in which UNIX honors these file system update requests.

The purpose of this document is to describe the normal updating of the file system, to discuss the possible causes of file system corruption, and to present the corrective actions implemented by *fsck*. Both the program and the interaction between the program and the operator are described.

†UNIX is a trademark of Bell Laboratories.

TABLE OF CONTENTS

1. Introduction

This document reflects the use of *fsck* with the Icon 4.2BSD file system organization. This is a revision of the original paper written by T. J. Kowalski.

When a UNIX operating system is brought up, a consistency check of the file systems should always be performed. This precautionary measure helps to insure a reliable environment for file storage on disk. If an inconsistency is discovered, corrective action must be taken. *Fsck* runs in two modes. Normally it is run non-interactively by the system after a normal boot. When running in this mode, it will only make changes to the file system that are known to always be correct. If an unexpected inconsistency is found *fsck* will exit with a non-zero exit status, leaving the system running single-user. Typically the operator then runs *fsck* interactively. When running in this mode, each problem is listed followed by a suggested corrective action. The operator must decide whether or not the suggested correction should be made. This second mode can be run by one of two methods. Either while running under UNIX or standalone. The standalone method allows recovering a filesystems when the filesystem is too corrupted to be used under UNIX.

The purpose of this memo is to dispel the mystique surrounding file system inconsistencies. It first describes the updating of the file system (the calm before the storm) and then describes file system corruption (the storm). Finally, the set of deterministic corrective actions used by *fsck* (the Coast Guard to the rescue) is presented.

2. Overview of the file system

The file system is discussed in detail in [Mckusick83]; this section gives a brief overview.

2.1. Superblock

A file system is described by its *super-block*. The super-block is built when the file system is created (*newfs*(8)) and never changes. The super-block contains the basic parameters of the file system, such as the number of data blocks it contains and a count of the maximum number of files. Because the super-block contains critical data, *newfs* replicates it to protect against catastrophic loss. The *default super block* always resides at a fixed offset from the beginning of the file system's disk partition. The *redundant super blocks* are not referenced unless a head crash or other hard disk error causes the default super-block to be unusable. The redundant blocks are sprinkled throughout the disk partition.

Within the file system are files. Certain files are distinguished as directories and contain collections of pointers to files that may themselves be directories. Every file has a descriptor associated with it called an *inode*. The inode contains information describing ownership of the file, time stamps indicating modification and access times for the file, and an array of indices pointing to the data blocks for the file. In this section, we assume that the first 12 blocks of the file are directly referenced by values stored in the inode structure itself†. The inode structure may also contain references to indirect blocks containing further data block indices. In a file system with a 4096 byte block size, a singly indirect block contains 1024 further block addresses, a doubly indirect block contains 1024 addresses of further single indirect blocks, and a triply indirect block contains 1024 addresses of further doubly indirect blocks.

In order to create files with up to 2↑32 bytes, using only two levels of indirection, the minimum size of a file system block is 4096 bytes. The size of file system blocks can be any power of two greater than or equal to 4096. The block size of the file system is maintained in the super-block, so it is possible for file systems of different block sizes to be accessible simultaneously on the same system. The block size must be decided when *newfs* creates the file system; the block size cannot be subsequently changed without rebuilding the file system.

2.2. Summary information

Associated with the super block is non replicated *summary information*. The summary information changes as the file system is modified. The summary information contains the number of blocks, fragments, inodes and directories in the file system.

2.3. Cylinder groups

The file system partitions the disk into one or more areas called *cylinder groups*. A cylinder group is comprised of one or more consecutive cylinders on a disk. Each cylinder group includes inode slots for files, a *block map* describing available blocks in the cylinder group, and summary information describing the usage of data blocks within the cylinder group. A fixed number of inodes is allocated for each cylinder group when the file system is created. The current policy is to allocate one inode for each 2048 bytes of disk space; this is expected to be far more inodes than will ever be needed.

All the cylinder group bookkeeping information could be placed at the beginning of each cylinder group. However if this approach were used, all the redundant information would be on the top platter. A single hardware failure that destroyed the top platter could cause the loss of all copies of the redundant super-blocks. Thus the cylinder group bookkeeping information begins at a floating offset from the beginning of the cylinder group. The offset for the *i+1*st cylinder group is about one track further from the beginning of the cylinder group than it was for the *i*th cylinder group. In this way, the redundant information spirals down into the pack; any single track, cylinder, or platter can be lost without losing all copies of the super-blocks. Except for the first

†The actual number may vary from system to system, but is usually in the range 5-13.

cylinder group, the space between the beginning of the cylinder group and the beginning of the cylinder group information stores data.

2.4. Fragments

To avoid waste in storing small files, the file system space allocator divides a single file system block into one or more *fragments*. The fragmentation of the file system is specified when the file system is created; each file system block can be optionally broken into 2, 4, or 8 addressable fragments. The lower bound on the size of these fragments is constrained by the disk sector size; typically 512 bytes is the lower bound on fragment size. The block map associated with each cylinder group records the space availability at the fragment level. Aligned fragments are examined to determine block availability.

On a file system with a block size of 4096 bytes and a fragment size of 1024 bytes, a file is represented by zero or more 4096 byte blocks of data, and possibly a single fragmented block. If a file system block must be fragmented to obtain space for a small amount of data, the remainder of the block is made available for allocation to other files. For example, consider an 11000 byte file stored on a 4096/1024 byte file system. This file uses two full size blocks and a 3072 byte fragment. If no fragments with at least 3072 bytes are available when the file is created, a full size block is split yielding the necessary 3072 byte fragment and an unused 1024 byte fragment. This remaining fragment can be allocated to another file, as needed.

2.5. Updates to the file system

Every working day hundreds of files are created, modified, and removed. Every time a file is modified, the operating system performs a series of file system updates. These updates, when written on disk, yield a consistent file system. The file system stages all modifications of critical information; modification can either be completed or cleanly backed out after a crash. Knowing the information that is first written to the file system, deterministic procedures can be developed to repair a corrupted file system. To understand this process, the order that the update requests were being honored must first be understood.

When a user program does an operation to change the file system, such as a *write*, the data to be written is copied into an internal *in-core* buffer in the kernel. Normally, the disk update is handled asynchronously; the user process is allowed to proceed even though the data has not yet been written to the disk. The data, along with the inode information reflecting the change, is eventually written out to disk. The real disk write may not happen until long after the *write* system call has returned. Thus at any given time, the file system, as it resides on the disk, lags the state of the file system represented by the in-core information.

The disk information is updated to reflect the in-core information when the buffer is required for another use, when a *sync*(2) is done (at 30 second intervals) by */etc/update*(8), or by manual operator intervention with the *sync*(8) command. If the system is halted without writing out the in-core information, the file system on the disk will be in an inconsistent state.

If all updates are done asynchronously, several serious inconsistencies can arise. One inconsistency is that a block may be claimed by two inodes. Such an inconsistency can occur when the system is halted before the pointer to the block in the old inode has been cleared in the copy of the old inode on the disk, and after the pointer to the block in the new inode has been written out to the copy of the new inode on the disk. Here, there is no deterministic method for deciding which inode should really claim the block. A similar problem can arise with a multiply claimed inode.

The problem with asynchronous inode updates can be avoided by doing all inode deallocations synchronously. Consequently, inodes and indirect blocks are written to the disk synchronously (*i.e.* the process blocks until the information is really written to disk) when they are being deallocated. Similarly inodes are kept consistent by synchronously deleting, adding, or changing directory entries.

3. Fixing corrupted file systems

A file system can become corrupted in several ways. The most common of these ways are improper shutdown procedures and hardware failures.

File systems may become corrupted during an *unclean halt*. This happens when proper shutdown procedures are not observed, physically write-protecting a mounted file system, or a mounted file system is taken off-line. The most common operator procedural failure is forgetting to *sync* the system before halting the CPU.

File systems may become further corrupted if proper startup procedures are not observed, e.g., not checking a file system for inconsistencies, and not repairing inconsistencies. Allowing a corrupted file system to be used (and, thus, to be modified further) can be disastrous.

Any piece of hardware can fail at any time. Failures can be as subtle as a bad block on a disk pack, or as blatant as a non-functional disk-controller.

3.1. Detecting and correcting corruption

Normally *fsck* is run non-interactively. In this mode it will only fix corruptions that are expected to occur from an unclean halt. These actions are a proper subset of the actions that *fsck* will take when it is running interactively. Throughout this paper we assume that *fsck* is being run interactively, and all possible errors can be encountered. When an inconsistency is discovered in this mode, *fsck* reports the inconsistency for the operator to chose a corrective action.

A quiescent‡ file system may be checked for structural integrity by performing consistency checks on the redundant data intrinsic to a file system. The redundant data is either read from the file system, or computed from other known values. The file system **must** be in a quiescent state when *fsck* is run, since *fsck* is a multi-pass program.

In the following sections, we discuss methods to discover inconsistencies and possible corrective actions for the cylinder group blocks, the inodes, the indirect blocks, and the data blocks containing directory entries.

3.2. Super-block checking

The most commonly corrupted item in a file system is the summary information associated with the super-block. The summary information is prone to corruption because it is modified with every change to the file system's blocks or inodes, and is usually corrupted after an unclean halt.

The super-block is checked for inconsistencies involving file-system size, number of inodes, free-block count, and the free-inode count. The file-system size must be larger than the number of blocks used by the super-block and the number of blocks used by the list of inodes. The file-system size and layout information are the most critical pieces of information for *fsck*. While there is no way to actually check these sizes, since they are statically determined by *newfs*, *fsck* can check that these sizes are within reasonable bounds. All other file system checks require that these sizes be correct. If *fsck* detects corruption in the static parameters of the default super-block, *fsck* requests the operator to specify the location of an alternate super-block.

3.3. Free block checking

Fsck checks that all the blocks marked as free in the cylinder group block maps are not claimed by any files. When all the blocks have been initially accounted for, *fsck* checks that the number of free blocks plus the number of blocks claimed by the inodes equals the total number of blocks in the file system.

If anything is wrong with the block allocation maps, *fsck* will rebuild them, based on the list it has computed of allocated blocks.

‡ I.e., unmounted and not being written on.

The summary information associated with the super-block counts the total number of free blocks within the file system. *Fsck* compares this count to the number of free blocks it found within the file system. If the two counts do not agree, then *fsck* replaces the incorrect count in the summary information by the actual free-block count.

The summary information counts the total number of free inodes within the file system. *Fsck* compares this count to the number of free inodes it found within the file system. If the two counts do not agree, then *fsck* replaces the incorrect count in the summary information by the actual free-inode count.

3.4. Checking the inode state

An individual inode is not as likely to be corrupted as the allocation information. However, because of the great number of active inodes, a few of the inodes are usually corrupted.

The list of inodes in the file system is checked sequentially starting with inode 2 (inode 0 marks unused inodes; inode 1 is saved for future generations) and progressing through the last inode in the file system. The state of each inode is checked for inconsistencies involving format and type, link count, duplicate blocks, bad blocks, and inode size.

Each inode contains a mode word. This mode word describes the type and state of the inode. Inodes must be one of six types: regular inode, directory inode, symbolic link inode, special block inode, special character inode, or socket inode. Inodes may be found in one of three allocation states: unallocated, allocated, and neither unallocated nor allocated. This last state suggests an incorrectly formated inode. An inode can get in this state if bad data is written into the inode list. The only possible corrective action is for *fsck* is to clear the inode.

3.5. Inode links

Each inode counts the total number of directory entries linked to the inode. *Fsck* verifies the link count of each inode by starting at the root of the file system, and descending through the directory structure. The actual link count for each inode is calculated during the descent.

If the stored link count is non-zero and the actual link count is zero, then no directory entry appears for the inode. If this happens, *fsck* will place the disconnected file in the *lost+found* directory. If the stored and actual link counts are non-zero and unequal, a directory entry may have been added or removed without the inode being updated. If this happens, *fsck* replaces the incorrect stored link count by the actual link count.

Each inode contains a list, or pointers to lists (indirect blocks), of all the blocks claimed by the inode. Since indirect blocks are owned by an inode, inconsistencies in indirect blocks directly affect the inode that owns it.

Fsck compares each block number claimed by an inode against a list of already allocated blocks. If another inode already claims a block number, then the block number is added to a list of *duplicate blocks*. Otherwise, the list of allocated blocks is updated to include the block number.

If there are any duplicate blocks, *fsck* will perform a partial second pass over the inode list to find the inode of the duplicated block. The second pass is needed, since without examining the files associated with these inodes for correct content, not enough information is available to determine which inode is corrupted and should be cleared. If this condition does arise (only hardware failure will cause it), then the inode with the earliest modify time is usually incorrect, and should be cleared. If this happens, *fsck* prompts the operator to clear both inodes. The operator must decide which one should be kept and which one should be cleared.

Fsck checks the range of each block number claimed by an inode. If the block number is lower than the first data block in the file system, or greater than the last data block, then the block number is a *bad block number*. Many bad blocks in an inode are usually caused by an indirect block that was not written to the file system, a condition which can only occur if there has been a hardware failure. If an inode contains bad block numbers, *fsck* prompts the operator to clear it.

3.6. Inode data size

Each inode contains a count of the number of data blocks that it contains. The number of actual data blocks is the sum of the allocated data blocks and the indirect blocks. *Fsck* computes the actual number of data blocks and compares that block count against the actual number of blocks the inode claims. If an inode contains an incorrect count *fsck* prompts the operator to fix it.

Each inode contains a thirty-two bit size field. The size is the number of data bytes in the file associated with the inode. The consistency of the byte size field is roughly checked by computing from the size field the maximum number of blocks that should be associated with the inode, and comparing that expected block count against the actual number of blocks the inode claims.

3.7. Checking the data associated with an inode

An inode can directly or indirectly reference three kinds of data blocks. All referenced blocks must be the same kind. The three types of data blocks are: plain data blocks, symbolic link data blocks, and directory data blocks. Plain data blocks contain the information stored in a file; symbolic link data blocks contain the path name stored in a link. Directory data blocks contain directory entries. *Fsck* can only check the validity of directory data blocks.

Each directory data block is checked for several types of inconsistencies. These inconsistencies include directory inode numbers pointing to unallocated inodes, directory inode numbers that are greater than the number of inodes in the file system, incorrect directory inode numbers for "." and "..", and directories that are not attached to the file system. If the inode number in a directory data block references an unallocated inode, then *fsck* will remove that directory entry. Again, this condition can only arise when there has been a hardware failure.

If a directory entry inode number references outside the inode list, then *fsck* will remove that directory entry. This condition occurs if bad data is written into a directory data block.

The directory inode number entry for "." must be the first entry in the directory data block. The inode number for "." must reference itself; e.g., it must equal the inode number for the directory data block. The directory inode number entry for ".." must be the second entry in the directory data block. Its value must equal the inode number for the parent of the directory entry (or the inode number of the directory data block if the directory is the root directory). If the directory inode numbers are incorrect, *fsck* will replace them with the correct values.

3.8. File system connectivity

Fsck checks the general connectivity of the file system. If directories are not linked into the file system, then *fsck* links the directory back into the file system in the *lost+found* directory. This condition only occurs when there has been a hardware failure.

Acknowledgements

I thank Bill Joy, Sam Leffler, Robert Elz and Dennis Ritchie for their suggestions and help in implementing the new file system. Thanks also to Robert Henry for his editorial input to get this document together. Finally we thank our sponsors, the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235. (Kirk McKusick, July 1983)

I would like to thank Larry A. Wehr for advice that lead to the first version of *fsck* and Rick B. Brandt for adapting *fsck* to UNIX/TS. (T. Kowalski, July 1979)

References

[Dolotta78] Dolotta, T. A., and Olsson, S. B. eds., *UNIX User's Manual, Edition 1.1* (January 1978).

[Joy83] Joy, W., Cooper, E., Fabry, R., Leffler, S., McKusick, M., and Mosher, D. *4.2BSD System Manual*, University of California at Berkeley, Computer Systems Research Group Technical Report #4, 1982.

[McKusick83] McKusick, M., Joy, W., Leffler, S., and Fabry, R. *A Fast File System for UNIX*, University of California at Berkeley, Computer Systems Research Group Technical Report #7, 1982.

[Ritchie78] Ritchie, D. M., and Thompson, K., The UNIX Time-Sharing System, *The Bell System Technical Journal* **57**, 6 (July-August 1978, Part 2), pp. 1905-29.

[Thompson78] Thompson, K., UNIX Implementation, *The Bell System Technical Journal* **57**, 6 (July-August 1978, Part 2), pp. 1931-46.

4. Appendix A – Fsck Error Conditions

4.1. Conventions

Fsck is a multi-pass file system check program. Each file system pass invokes a different Phase of the *fsck* program. After the initial setup, *fsck* performs successive Phases over each file system, checking blocks and sizes, path-names, connectivity, reference counts, and the map of free blocks, (possibly rebuilding it), and performs some cleanup.

Normally *fsck* is run non-interactively to *preen* the file systems after an unclean halt. While preen'ing a file system, it will only fix corruptions that are expected to occur from an unclean halt. These actions are a proper subset of the actions that *fsck* will take when it is running interactively. Throughout this appendix many errors have several options that the operator can take. When an inconsistency is detected, *fsck* reports the error condition to the operator. If a response is required, *fsck* prints a prompt message and waits for a response. When preen'ing most errors are fatal. For those that are expected, the response taken is noted. This appendix explains the meaning of each error condition, the possible responses, and the related error conditions.

The error conditions are organized by the *Phase* of the *fsck* program in which they can occur. The error conditions that may occur in more than one Phase will be discussed in initialization.

4.2. Initialization

Before a file system check can be performed, certain tables have to be set up and certain files opened. This section concerns itself with the opening of files and the initialization of tables. This section lists error conditions resulting from command line options, memory requests, opening of files, status of files, file system size checks, and creation of the scratch file. All of the initialization errors are fatal when the file system is being preen'ed.

C **option?**
C is not a legal option to *fsck*; legal options are –b, –y, –n, and –p. *Fsck* terminates on this error condition. See the *fsck*(8) manual entry for further detail.

cannot alloc NNN bytes for blockmap
cannot alloc NNN bytes for freemap
cannot alloc NNN bytes for statemap
cannot alloc NNN bytes for lncntp
Fsck's request for memory for its virtual memory tables failed. This should never happen. *Fsck* terminates on this error condition. See a guru.

Can't open checklist file: *F*
The file system checklist file *F* (usually */etc/fstab*) can not be opened for reading. *Fsck* terminates on this error condition. Check access modes of *F*.

Can't stat root
Fsck's request for statistics about the root directory "/" failed. This should never happen. *Fsck* terminates on this error condition. See a guru.

Can't stat *F*
Can't make sense out of name *F*
Fsck's request for statistics about the file system *F* failed. When running manually, it ignores this file system and continues checking the next file system given. Check access modes of *F*.

Can't open *F*
Fsck's request attempt to open the file system *F* failed. When running manually, it ignores this file system and continues checking the next file system given. Check access modes of *F*.

F: **(NO WRITE)**
Either the –n flag was specified or *fsck*'s attempt to open the file system *F* for writing failed. When running manually, all the diagnostics are printed out, but no modifications are attempted to fix them.

file is not a block or character device; OK
You have given *fsck* a regular file name by mistake. Check the type of the file specified.

Possible responses to the OK prompt are:

YES Ignore this error condition.

NO ignore this file system and continues checking the next file system given.

One of the following messages will appear:
MAGIC NUMBER WRONG
NCG OUT OF RANGE
CPG OUT OF RANGE
NCYL DOES NOT JIVE WITH NCG*CPG
SIZE PREPOSTEROUSLY LARGE
TRASHED VALUES IN SUPER BLOCK

and will be followed by the message:
F: **BAD SUPER BLOCK:** *B*
USE -b OPTION TO FSCK TO SPECIFY LOCATION OF AN ALTERNATE SUPER-BLOCK TO SUPPLY NEEDED INFORMATION; SEE fsck(8).
The super block has been corrupted. An alternative super block must be selected from among those listed by *newfs* (8) when the file system was created. For file systems with a blocksize less than 32K, specifying –b 32 is a good first choice.

INTERNAL INCONSISTENCY: *M*
Fsck's has had an internal panic, whose message is specified as *M*. This should never happen. See a guru.

CAN NOT SEEK: BLK *B* **(CONTINUE)**
Fsck's request for moving to a specified block number *B* in the file system failed. This should never happen. See a guru.

Possible responses to the CONTINUE prompt are:

YES attempt to continue to run the file system check. Often, however the problem will persist. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If the block was part of the virtual memory buffer cache, *fsck* will terminate with the message "Fatal I/O error".

NO terminate the program.

CAN NOT READ: BLK *B* **(CONTINUE)**
Fsck's request for reading a specified block number *B* in the file system failed. This should never happen. See a guru.

Possible responses to the CONTINUE prompt are:

YES attempt to continue to run the file system check. Often, however, the problem will persist. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If the block was part of the virtual memory buffer cache, *fsck* will terminate with the message "Fatal I/O error".

NO terminate the program.

CAN NOT WRITE: BLK *B* (CONTINUE)
Fsck's request for writing a specified block number *B* in the file system failed. The disk is write-protected. See a guru.

Possible responses to the CONTINUE prompt are:

YES attempt to continue to run the file system check. Often, however, the problem will persist. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If the block was part of the virtual memory buffer cache, *fsck* will terminate with the message "Fatal I/O error".

NO terminate the program.

4.3. Phase 1 – Check Blocks and Sizes

This phase concerns itself with the inode list. This section lists error conditions resulting from checking inode types, setting up the zero-link-count table, examining inode block numbers for bad or duplicate blocks, checking inode size, and checking inode format. All errors in this phase except **INCORRECT BLOCK COUNT** are fatal if the file system is being preen'ed,

CG *C*: BAD MAGIC NUMBER The magic number of cylinder group *C* is wrong. This usually indicates that the cylinder group maps have been destroyed. When running manually the cylinder group is marked as needing to be reconstructed.

UNKNOWN FILE TYPE I=*I* (CLEAR) The mode word of the inode *I* indicates that the inode is not a special block inode, special character inode, socket inode, regular inode, symbolic link, or directory inode.

Possible responses to the CLEAR prompt are:

YES de-allocate inode *I* by zeroing its contents. This will always invoke the UNALLOCATED error condition in Phase 2 for each directory entry pointing to this inode.

NO ignore this error condition.

LINK COUNT TABLE OVERFLOW (CONTINUE)
An internal table for *fsck* containing allocated inodes with a link count of zero has no more room. Recompile *fsck* with a larger value of MAXLNCNT.

Possible responses to the CONTINUE prompt are:

YES continue with the program. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If another allocated inode with a zero link count is found, this error condition is repeated.

NO terminate the program.

B* BAD I=*I
Inode *I* contains block number *B* with a number lower than the number of the first data block in the file system or greater than the number of the last block in the file system. This error condition may invoke the **EXCESSIVE BAD BLKS** error condition in Phase 1 if inode *I* has too many block numbers outside the file system range. This error condition will always invoke the **BAD/DUP** error condition in Phase 2 and Phase 4.

EXCESSIVE BAD BLKS I=*I* (CONTINUE)
There is more than a tolerable number (usually 10) of blocks with a number lower than the number of the first data block in the file system or greater than the number of last block in the file system associated with inode *I*.

Possible responses to the CONTINUE prompt are:

YES ignore the rest of the blocks in this inode and continue checking with the next inode in the file system. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system.

NO terminate the program.

B **DUP I=***I*

Inode *I* contains block number *B* which is already claimed by another inode. This error condition may invoke the **EXCESSIVE DUP BLKS** error condition in Phase 1 if inode *I* has too many block numbers claimed by other inodes. This error condition will always invoke Phase 1b and the **BAD/DUP** error condition in Phase 2 and Phase 4.

EXCESSIVE DUP BLKS I=*I* **(CONTINUE)**

There is more than a tolerable number (usually 10) of blocks claimed by other inodes.

Possible responses to the CONTINUE prompt are:

YES ignore the rest of the blocks in this inode and continue checking with the next inode in the file system. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system.

NO terminate the program.

DUP TABLE OVERFLOW (CONTINUE)

An internal table in *fsck* containing duplicate block numbers has no more room. Recompile *fsck* with a larger value of DUPTBLSIZE.

Possible responses to the CONTINUE prompt are:

YES continue with the program. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If another duplicate block is found, this error condition will repeat.

NO terminate the program.

PARTIALLY ALLOCATED INODE I=*I* **(CLEAR)**

Inode *I* is neither allocated nor unallocated.

Possible responses to the CLEAR prompt are:

YES de-allocate inode *I* by zeroing its contents.

NO ignore this error condition.

INCORRECT BLOCK COUNT I=*I* **(***X* **should be** *Y*) **(CORRECT)**

The block count for inode *I* is *X* blocks, but should be *Y* blocks. When preen'ing the count is corrected.

Possible responses to the CORRECT prompt are:

YES replace the block count of inode *I* with *Y*.

NO ignore this error condition.

4.4. Phase 1B: Rescan for More Dups

When a duplicate block is found in the file system, the file system is rescanned to find the inode which previously claimed that block. This section lists the error condition when the duplicate block is found.

B **DUP I=***I*
Inode *I* contains block number *B* that is already claimed by another inode. This error condition will always invoke the **BAD/DUP** error condition in Phase 2. You can determine which inodes have overlapping blocks by examining this error condition and the DUP error condition in Phase 1.

4.5. Phase 2 – Check Pathnames

This phase concerns itself with removing directory entries pointing to error conditioned inodes from Phase 1 and Phase 1b. This section lists error conditions resulting from root inode mode and status, directory inode pointers in range, and directory entries pointing to bad inodes. All errors in this phase are fatal if the file system is being preen'ed.

ROOT INODE UNALLOCATED. TERMINATING.
The root inode (usually inode number 2) has no allocate mode bits. This should never happen. The program will terminate.

NAME TOO LONG *F*
An excessively long path name has been found. This is usually indicative of loops in the file system name space. This can occur if the super user has made circular links to directories. The offending links must be removed (by a guru).

ROOT INODE NOT DIRECTORY (FIX)
The root inode (usually inode number 2) is not directory inode type.

Possible responses to the FIX prompt are:

YES replace the root inode's type to be a directory. If the root inode's data blocks are not directory blocks, a VERY large number of error conditions will be produced.

NO terminate the program.

DUPS/BAD IN ROOT INODE (CONTINUE)
Phase 1 or Phase 1b have found duplicate blocks or bad blocks in the root inode (usually inode number 2) for the file system.

Possible responses to the CONTINUE prompt are:

YES ignore the **DUPS/BAD** error condition in the root inode and attempt to continue to run the file system check. If the root inode is not correct, then this may result in a large number of other error conditions.

NO terminate the program.

I OUT OF RANGE I=*I* **NAME=***F* **(REMOVE)**
A directory entry *F* has an inode number *I* which is greater than the end of the inode list.

Possible responses to the REMOVE prompt are:

YES the directory entry *F* is removed.

NO ignore this error condition.

UNALLOCATED I=*I* **OWNER=***O* **MODE=***M* **SIZE=***S* **MTIME=***T* **DIR=***F* **(REMOVE)**
A directory entry *F* has a directory inode *I* without allocate mode bits. The owner *O*, mode *M*, size *S*, modify time *T*, and directory name *F* are printed.

Possible responses to the REMOVE prompt are:

YES the directory entry *F* is removed.

NO ignore this error condition.

UNALLOCATED I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* FILE=*F* (REMOVE)
A directory entry *F* has an inode *I* without allocate mode bits. The owner *O*, mode *M*, size *S*, modify time *T*, and file name *F* are printed.

Possible responses to the REMOVE prompt are:

YES the directory entry *F* is removed.

NO ignore this error condition.

DUP/BAD I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (REMOVE)
Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with directory entry *F*, directory inode *I*. The owner *O*, mode *M*, size *S*, modify time *T*, and directory name *F* are printed.

Possible responses to the REMOVE prompt are:

YES the directory entry *F* is removed.

NO ignore this error condition.

DUP/BAD I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* FILE=*F* (REMOVE)
Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with directory entry *F*, inode *I*. The owner *O*, mode *M*, size *S*, modify time *T*, and file name *F* are printed.

Possible responses to the REMOVE prompt are:

YES the directory entry *F* is removed.

NO ignore this error condition.

ZERO LENGTH DIRECTORY I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (REMOVE)
A directory entry *F* has a size *S* that is zero. The owner *O*, mode *M*, size *S*, modify time *T*, and directory name *F* are printed.

Possible responses to the REMOVE prompt are:

YES the directory entry *F* is removed; this will always invoke the BAD/DUP error condition in Phase 4.

NO ignore this error condition.

DIRECTORY TOO SHORT I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *F* has been found whose size *S* is less than the minimum size directory. The owner *O*, mode *M*, size *S*, modify time *T*, and directory name *F* are printed.

Possible responses to the FIX prompt are:

YES increase the size of the directory to the minimum directory size.

NO ignore this directory.

DIRECTORY CORRUPTED I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (SALVAGE)
A directory with an inconsistent internal state has been found.

Possible responses to the FIX prompt are:

YES throw away all entries up to the next 512-byte boundary. This rather drastic action can throw away up to 42 entries, and should be taken only after other recovery efforts have

failed.

NO Skip up to the next 512-byte boundary and resume reading, but do not modify the directory.

BAD INODE NUMBER FOR '.' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *I* has been found whose inode number for '.' does does not equal *I*.

Possible responses to the FIX prompt are:

YES change the inode number for '.' to be equal to *I*.

NO leave the inode number for '.' unchanged.

MISSING '.' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *I* has been found whose first entry is unallocated.

Possible responses to the FIX prompt are:

YES make an entry for '.' with inode number equal to *I*.

NO leave the directory unchanged.

MISSING '.' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F*
CANNOT FIX, FIRST ENTRY IN DIRECTORY CONTAINS *F*
A directory *I* has been found whose first entry is *F*. *Fsck* cannot resolve this problem. The file system should be mounted and the offending entry *F* moved elsewhere. The file system should then be unmounted and *fsck* should be run again.

MISSING '.' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F*
CANNOT FIX, INSUFFICIENT SPACE TO ADD '.'
A directory *I* has been found whose first entry is not '.'. *Fsck* cannot resolve this problem as it should never happen. See a guru.

EXTRA '.' ENTRY I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *I* has been found that has more than one entry for '.'.

Possible responses to the FIX prompt are:

YES remove the extra entry for '.'.

NO leave the directory unchanged.

BAD INODE NUMBER FOR '..' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *I* has been found whose inode number for '..' does does not equal the parent of *I*.

Possible responses to the FIX prompt are:

YES change the inode number for '..' to be equal to the parent of *I*.

NO leave the inode number for '..' unchanged.

MISSING '..' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *I* has been found whose second entry is unallocated.

Possible responses to the FIX prompt are:

YES make an entry for '..' with inode number equal to the parent of *I*.

NO leave the directory unchanged.

MISSING '..' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F*
CANNOT FIX, SECOND ENTRY IN DIRECTORY CONTAINS *F*
A directory *I* has been found whose second entry is *F*. *Fsck* cannot resolve this problem. The file system should be mounted and the offending entry *F* moved elsewhere. The file system should then be unmounted and *fsck* should be run again.

MISSING '..' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F*
CANNOT FIX, INSUFFICIENT SPACE TO ADD '..'
A directory *I* has been found whose second entry is not '..'. *Fsck* cannot resolve this problem as it should never happen. See a guru.

EXTRA '..' ENTRY I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *I* has been found that has more than one entry for '..'.

Possible responses to the FIX prompt are:

YES remove the extra entry for '..'.

NO leave the directory unchanged.

4.6. Phase 3 – Check Connectivity

This phase concerns itself with the directory connectivity seen in Phase 2. This section lists error conditions resulting from unreferenced directories, and missing or full *lost+found* directories.

UNREF DIR I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (RECONNECT)
The directory inode *I* was not connected to a directory entry when the file system was traversed. The owner *O*, mode *M*, size *S*, and modify time *T* of directory inode *I* are printed. When preen'ing, the directory is reconnected if its size is non-zero, otherwise it is cleared.

Possible responses to the RECONNECT prompt are:

YES reconnect directory inode *I* to the file system in the directory for lost files (usually *lost+found*). This may invoke the *lost+found* error condition in Phase 3 if there are problems connecting directory inode *I* to *lost+found*. This may also invoke the CONNECTED error condition in Phase 3 if the link was successful.

NO ignore this error condition. This will always invoke the UNREF error condition in Phase 4.

SORRY. NO lost+found DIRECTORY
There is no *lost+found* directory in the root directory of the file system; *fsck* ignores the request to link a directory in *lost+found*. This will always invoke the UNREF error condition in Phase 4. Check access modes of *lost+found*. See *fsck*(8) manual entry for further detail. This error is fatal if the file system is being preen'ed.

SORRY. NO SPACE IN lost+found DIRECTORY
There is no space to add another entry to the *lost+found* directory in the root directory of the file system; *fsck* ignores the request to link a directory in *lost+found*. This will always invoke the UNREF error condition in Phase 4. Clean out unnecessary entries in *lost+found* or make *lost+found* larger. See *fsck*(8) manual entry for further detail. This error is fatal if the file system is being preen'ed.

DIR I=*I1* CONNECTED. PARENT WAS I=*I2*
This is an advisory message indicating a directory inode *I1* was successfully connected to the *lost+found* directory. The parent inode *I2* of the directory inode *I1* is replaced by the inode number of the *lost+found* directory.

4.7. Phase 4 – Check Reference Counts

This phase concerns itself with the link count information seen in Phase 2 and Phase 3. This section lists error conditions resulting from unreferenced files, missing or full *lost+found* directory, incorrect link counts for files, directories, symbolic links, or special files, unreferenced files, symbolic links, and directories, bad and duplicate blocks in files, symbolic links, and directories, and incorrect total free-inode counts. All errors in this phase are correctable if the file system is being preen'ed except running out of space in the lost+found directory.

UNREF FILE I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (RECONNECT)
Inode *I* was not connected to a directory entry when the file system was traversed. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed. When preen'ing the file is cleared if either its size or its link count is zero, otherwise it is reconnected.

Possible responses to the RECONNECT prompt are:

YES reconnect inode *I* to the file system in the directory for lost files (usually *lost+found*). This may invoke the *lost+found* error condition in Phase 4 if there are problems connecting inode *I* to *lost+found*.

NO ignore this error condition. This will always invoke the CLEAR error condition in Phase 4.

(CLEAR)
The inode mentioned in the immediately previous error condition can not be reconnected. This cannot occur if the file system is being preen'ed, since lack of space to reconnect files is a fatal error.

Possible responses to the CLEAR prompt are:

YES de-allocate the inode mentioned in the immediately previous error condition by zeroing its contents.

NO ignore this error condition.

SORRY. NO lost+found DIRECTORY
There is no *lost+found* directory in the root directory of the file system; *fsck* ignores the request to link a file in *lost+found*. This will always invoke the CLEAR error condition in Phase 4. Check access modes of *lost+found*. This error is fatal if the file system is being preen'ed.

SORRY. NO SPACE IN lost+found DIRECTORY
There is no space to add another entry to the *lost+found* directory in the root directory of the file system; *fsck* ignores the request to link a file in *lost+found*. This will always invoke the CLEAR error condition in Phase 4. Check size and contents of *lost+found*. This error is fatal if the file system is being preen'ed.

LINK COUNT FILE I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* COUNT=*X* SHOULD BE *Y* (ADJUST)
The link count for inode *I* which is a file, is *X* but should be *Y*. The owner *O*, mode *M*, size *S*, and modify time *T* are printed. When preen'ing the link count is adjusted.

Possible responses to the ADJUST prompt are:

YES replace the link count of file inode *I* with *Y*.

NO ignore this error condition.

LINK COUNT DIR I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* COUNT=*X* SHOULD BE *Y* (ADJUST)
The link count for inode *I* which is a directory, is *X* but should be *Y*. The owner *O*, mode *M*, size *S*, and modify time *T* of directory inode *I* are printed. When preen'ing the link count is adjusted.

Possible responses to the ADJUST prompt are:

YES replace the link count of directory inode *I* with *Y*.

NO ignore this error condition.

LINK COUNT *F* I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* COUNT=*X* SHOULD BE *Y* (ADJUST)

The link count for *F* inode *I* is *X* but should be *Y*. The name *F*, owner *O*, mode *M*, size *S*, and modify time *T* are printed. When preen'ing the link count is adjusted.

Possible responses to the ADJUST prompt are:

YES replace the link count of inode *I* with *Y*.

NO ignore this error condition.

UNREF FILE I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (CLEAR)

Inode *I* which is a file, was not connected to a directory entry when the file system was traversed. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed. When preen'ing, this is a file that was not connected because its size or link count was zero, hence it is cleared.

Possible responses to the CLEAR prompt are:

YES de-allocate inode *I* by zeroing its contents.

NO ignore this error condition.

UNREF DIR I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (CLEAR)

Inode *I* which is a directory, was not connected to a directory entry when the file system was traversed. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed. When preen'ing, this is a directory that was not connected because its size or link count was zero, hence it is cleared.

Possible responses to the CLEAR prompt are:

YES de-allocate inode *I* by zeroing its contents.

NO ignore this error condition.

BAD/DUP FILE I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (CLEAR)

Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with file inode *I*. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed. This error cannot arise when the file system is being preen'ed, as it would have caused a fatal error earlier.

Possible responses to the CLEAR prompt are:

YES de-allocate inode *I* by zeroing its contents.

NO ignore this error condition.

BAD/DUP DIR I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (CLEAR)

Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with directory inode *I*. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed. This error cannot arise when the file system is being preen'ed, as it would have caused a fatal error earlier.

Possible responses to the CLEAR prompt are:

YES de-allocate inode *I* by zeroing its contents.

NO ignore this error condition.

FREE INODE COUNT WRONG IN SUPERBLK (FIX)

The actual count of the free inodes does not match the count in the super-block of the file system. When preen'ing, the count is fixed.

Possible responses to the FIX prompt are:

YES replace the count in the super-block by the actual count.

NO ignore this error condition.

4.8. Phase 5 - Check Cyl groups

This phase concerns itself with the free-block maps. This section lists error conditions resulting from allocated blocks in the free-block maps, free blocks missing from free-block maps, and the total free-block count incorrect.

CG *C*: BAD MAGIC NUMBER
The magic number of cylinder group *C* is wrong. This usually indicates that the cylinder group maps have been destroyed. When running manually the cylinder group is marked as needing to be reconstructed. This error is fatal if the file system is being preen'ed.

EXCESSIVE BAD BLKS IN BIT MAPS (CONTINUE)
An inode contains more than a tolerable number (usually 10) of blocks claimed by other inodes or that are out of the legal range for the file system. This error is fatal if the file system is being preen'ed.

Possible responses to the CONTINUE prompt are:

YES ignore the rest of the free-block maps and continue the execution of *fsck*.

NO terminate the program.

SUMMARY INFORMATION *T* BAD
where *T* is one or more of:
(INODE FREE)
(BLOCK OFFSETS)
(FRAG SUMMARIES)
(SUPER BLOCK SUMMARIES)
The indicated summary information was found to be incorrect. This error condition will always invoke the **BAD CYLINDER GROUPS** condition in Phase 6. When preen'ing, the summary information is recomputed.

***X* BLK(S) MISSING**
X blocks unused by the file system were not found in the free-block maps. This error condition will always invoke the **BAD CYLINDER GROUPS** condition in Phase 6. When preen'ing, the block maps are rebuilt.

BAD CYLINDER GROUPS (SALVAGE)
Phase 5 has found bad blocks in the free-block maps, duplicate blocks in the free-block maps, or blocks missing from the file system. When preen'ing, the cylinder groups are reconstructed.

Possible responses to the SALVAGE prompt are:

YES replace the actual free-block maps with a new free-block maps.

NO ignore this error condition.

FREE BLK COUNT WRONG IN SUPERBLOCK (FIX)
The actual count of free blocks does not match the count in the super-block of the file system. When preen'ing, the counts are fixed.

Possible responses to the FIX prompt are:

YES replace the count in the super-block by the actual count.

NO ignore this error condition.

4.9. Phase 6 - Salvage Cylinder Groups

This phase concerns itself with the free-block maps reconstruction. No error messages are produced.

4.10. Cleanup

Once a file system has been checked, a few cleanup functions are performed. This section lists advisory messages about the file system and modify status of the file system.

***V* files, *W* used, *X* free (*Y* frags, *Z* blocks)**

This is an advisory message indicating that the file system checked contained *V* files using *W* fragment sized blocks leaving *X* fragment sized blocks free in the file system. The numbers in parenthesis breaks the free count down into *Y* free fragments and *Z* free full sized blocks.

******* REBOOT UNIX *******

This is an advisory message indicating that the root file system has been modified by *fsck*. If UNIX is not rebooted immediately, the work done by *fsck* may be undone by the in-core copies of tables UNIX keeps. When preen'ing, *fsck* will exit with a code of 4. The auto-reboot script interprets an exit code of 4 by issuing a reboot system call.

******* FILE SYSTEM WAS MODIFIED *******

This is an advisory message indicating that the current file system was modified by *fsck*. If this file system is mounted or is the current root file system, *fsck* should be halted and UNIX rebooted. If UNIX is not rebooted immediately, the work done by *fsck* may be undone by the in-core copies of tables UNIX keeps.

4.2BSD Line Printer Spooler Manual

Revised July 27, 1983

Ralph Campbell

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

(415) 642-7780

ABSTRACT

This document describes the structure and installation procedure for the line printer spooling system developed for the 4.2BSD version of the UNIX* operating system.

1. Overview

The line printer system supports:

- multiple printers,
- multiple spooling queues,
- both local and remote printers, and
- printers attached via serial lines which require line initialization such as the baud rate.

Raster output devices such as a Varian or Versatec, and laser printers such as an Imagen, are also supported by the line printer system.

The line printer system consists mainly of the following files and commands:

/etc/printcap	printer configuration and capability data base
/usr/lib/lpd	line printer daemon, does all the real work
/usr/ucb/lpr	program to enter a job in a printer queue
/usr/ucb/lpq	spooling queue examination program
/usr/ucb/lprm	program to delete jobs from a queue
/etc/lpc	program to administer printers and spooling queues
/dev/printer	socket on which lpd listens

The file /etc/printcap is a master data base describing line printers directly attached to a machine and, also, printers accessible across a network. The manual page entry *printcap*(5) provides the ultimate definition of the format of this data base, as well as indicating default values for important items such as the directory in which spooling is performed. This document highlights the important information which may be placed *printcap*.

---

* UNIX is a trademark of Bell Laboratories.

2. Commands

2.1. lpd – line printer dameon

The program *lpd*(8), usually invoked at boot time from the /etc/rc file, acts as a master server for coordinating and controlling the spooling queues configured in the printcap file. When *lpd* is started it makes a single pass through the *printcap* database restarting any printers which have jobs. In normal operation *lpd* listens for service requests on multiple sockets, one in the UNIX domain (named "/dev/printer") for local requests, and one in the Internet domain (under the "printer" service specification) for requests for printer access from off machine; see *socket*(2) and *services*(5) for more information on sockets and service specifications, respectively. *Lpd* spawns a copy of itself to process the request; the master daemon continues to listen for new requests.

Clients communicate with *lpd* using a simple transaction oriented protocol. Authentication of remote clients is done based on the "privilege port" scheme employed by *rshd*(8C) and *rcmd*(3X). The following table shows the requests understood by *lpd*. In each request the first byte indicates the "meaning" of the request, followed by the name of the printer to which it should be applied. Additional qualifiers may follow, depending on the request.

Request	Interpretation
^Aprinter\n	check the queue for jobs and print any found
^Bprinter\n	receive and queue a job from another machine
^Cprinter [users ...] [jobs ...]\n	return short list of current queue state
^Dprinter [users ...] [jobs ...]\n	return long list of current queue state
^Eprinter person [users ...] [jobs ...]\n	remove jobs from a queue

The *lpr*(1) command is used by users to enter a print job in a local queue and to notify the local *lpd* that there are new jobs in the spooling area. *Lpd* either schedules the job to be printed locally, or in the case of remote printing, attempts to forward the job to the appropriate machine. If the printer cannot be opened or the destination machine is unreachable, the job will remain queued until it is possible to complete the work.

2.2. lpq – show line printer queue

The *lpq*(1) program works recursively backwards displaying the queue of the machine with the printer and then the queue(s) of the machine(s) that lead to it. *Lpq* has two forms of output: in the default, short, format it gives a single line of output per queued job; in the long format it shows the list of files, and their sizes, which comprise a job.

2.3. lprm – remove jobs from a queue

The *lprm*(1) command deletes jobs from a spooling queue. If necessary, *lprm* will first kill off a running daemon which is servicing the queue, restarting it after the required files are removed. When removing jobs destined for a remote printer, *lprm* acts similarly to *lpq* except it first checks locally for jobs to remove and then tries to remove files in queues off-machine.

2.4. lpc – line printer control program

The *lpc*(8) program is used by the system administrator to control the operation of the line printer system. For each line printer configured in /etc/printcap, *lpc* may be used to:

- disable or enable a printer,
- disable or enable a printer's spooling queue,
- rearrange the order of jobs in a spooling queue,
- find the status of printers, and their associated spooling queues and printer dameons.

3. Access control

The printer system maintains protected spooling areas so that users cannot circumvent printer accounting or remove files other than their own. The strategy used to maintain protected spooling areas is as follows:

- The spooling area is writable only by a *daemon* user and *spooling* group.
- The *lpr* program runs setuid *root* and setgid *spooling*. The *root* access is used to read any file required, verifying accessibility with an *access*(2) call. The group ID is used in setting up proper ownership of files in the spooling area for *lprm*.
- Control files in a spooling area are made with *daemon* ownership and group ownership *spooling*. Their mode is 0660. This insures control files are not modified by a user and that no user can remove files except through *lprm*.
- The spooling programs, *lpd*, *lpq*, and *lprm* run setuid *root* and setgid *spooling* to access spool files and printers.
- The printer server, *lpd*, uses the same verification procedures as *rshd*(8C) in authenticating remote clients. The host on which a client resides must be present in the file /etc/hosts.equiv, used to create clusters of machines under a single administration.

In practice, none of *lpd*, *lpq*, or *lprm* would have to run as user *root* if remote spooling were not supported. In previous incarnations of the printer system *lpd* ran setuid *daemon*, setgid *spooling*, and *lpq* and *lprm* ran setgid *spooling*.

4. Setting up

The 4.2BSD release comes with the necessary programs installed and with the default line printer queue created. If the system must be modified, the makefile in the directory /usr/src/usr.lib/lpr should be used in recompiling and reinstalling the necessary programs.

The real work in setting up is to create the *printcap* file and any printer filters for printers not supported in the distribution system.

4.1. Creating a printcap file

The *printcap* database contains one or more entries per printer. A printer should have a separate spooling directory; otherwise, jobs will be printed on different printers depending on which printer daemon starts first. This section describes how to create entries for printers which do not conform to the default printer description (an LP-11 style interface to a standard, band printer).

4.1.1. Printers on serial lines

When a printer is connected via a serial communication line it must have the proper baud rate and terminal modes set. The following example is for a DecWriter III printer connected locally via a 1200 baud serial line.

```
lp|LA-180 DecWriter III:\
        :lp=/dev/lp:br#1200:fs#06320:\
        :tr=\f:of=/usr/lib/lpf:lf=/usr/adm/lpd-errs:
```

The **lp** entry specifies the file name to open for output. In this case it could be left out since "/dev/lp" is the default. The **br** entry sets the baud rate for the tty line and the **fs** entry sets CRMOD, no parity, and XTABS (see *tty*(4)). The **tr** entry indicates a form-feed should be printed when the queue empties so the paper can be torn off without turning the printer off-line and pressing form feed. The **of** entry specifies the filter program *lpf* should be used for printing the files; more will be said about filters later. The last entry causes errors to be written to the file "/usr/adm/lpd-errs" instead of the console.

4.1.2. Remote printers

Printers which reside on remote hosts should have an empty **lp** entry. For example, the following printcap entry would send output to the printer named "lp" on the machine "ucbvax".

```
lp|default line printer:\
        :lp=:rm=ucbvax:rp=lp:sd=/usr/spool/vaxlpd:
```

The **rm** entry is the name of the remote machine to connect to; this name must appear in the /etc/hosts database, see *hosts*(5). The **rp** capability indicates the name of the printer on the remote machine is "lp"; in this case it could be left out since this is the default value. The **sd** entry specifies "/usr/spool/vaxlpd" as the spooling directory instead of the default value of "/usr/spool/lpd".

4.2. Output filters

Filters are used to handle device dependencies and to perform accounting functions. The output filter **of** is used to filter text data to the printer device when accounting is not used or when all text data must be passed through a filter. It is not intended to perform accounting since it is started only once, all text files are filtered through it, and no provision is made for passing owners' login name, identifying the begining and ending of jobs, etc. The other filters (if specified) are started for each file printed and perform accounting if there is an **af** entry. If entries for both **of** and one of the other filters are specified, the output filter is used only to print the banner page; it is then stopped to allow other filters access to the printer. An example of a printer which requires output filters is the Benson-Varian.

```
va|varian|Benson-Varian:\
        :lp=/dev/va0:sd=/usr/spool/vad:of=/usr/lib/vpf:\
        :tf=/usr/lib/rvcat:mx#2000:pl#58:tr=\f:
```

The **tf** entry specifies "/usr/lib/rvcat" as the filter to be used in printing *troff*(1) output. This filter is needed to set the device into print mode for text, and plot mode for printing *troff* files and raster images (see *va*(4V)). Note that the page length is set to 58 lines by the **pl** entry for 8.5" by 11" fan-fold paper. To enable accounting, the varian entry would be augmented with an **af** filter as shown below.

```
va|varian|Benson-Varian:\
        :lp=/dev/va0:sd=/usr/spool/vad:of=/usr/lib/vpf:\
        :if=/usr/lib/vpf:tf=/usr/lib/rvcat:af=/usr/adm/vaacct:\
        :mx#2000:pl#58:tr=\f:
```

5. Output filter specifications

The filters supplied with 4.2BSD handle printing and accounting for most common line printers, the Benson-Varian, the wide (36") and narrow (11") Versatec printer/plotters. For other devices or accounting methods, it may be necessary to create a new filter.

Filters are spawned by *lpd* with their standard input the data to be printed, and standard output the printer. The standard error is attached to the **lf** file for logging errors. A filter must return a 0 exit code if there were no errors, 1 if the job should be reprinted, and 2 if the job should be thrown away. When *lprm* sends a kill signal to the *lpd* process controlling printing, it sends a SIGINT signal to all filters and descendents of filters. This signal can be trapped by filters which need to perform cleanup operations such as deleting temporary files.

Arguments passed to a filter depend on its type. The **of** filter is called with the following arguments.

ofiler –**w**width –**l**length

The *width* and *length* values come from the **pw** and **pl** entries in the printcap database. The **if** filter is passed the following parameters.

filter [**–c**] **–w**width **–l**length **–i**indent **–n** login **–h** host accounting_file

The **–c** flag is optional, and only supplied when control characters are to be passed uninterpreted to the printer (when the **–l** option of *lpr* is used to print the file). The **–w** and **–l** parameters are the same as for the **of** filter. The **–n** and **–h** parameters specify the login name and host name of the job owner. The last argument is the name of the accounting file from *printcap*.

All other filters are called with the following arguments:

filter **–x**width **–y**length **–n** login **–h** host accounting_file

The **–x** and **–y** options specify the horizontal and vertical page size in pixels (from the **px** and **py** entries in the printcap file). The rest of the arguments are the same as for the **if** filter.

6. Line printer Administration

The *lpc* program provides local control over line printer activity. The major commands and their intended use will be described. The command format and remaining commands are described in *lpc*(8).

abort and **start**

Abort terminates an active spooling daemon on the local host immediately and then disables printing (preventing new daemons from being started by *lpr*). This is normally used to forciblly restart a hung line printer daemon (i.e., *lpq* reports that there is a daemon present but nothing is happening). It does not remove any jobs from the queue (use the *lprm* command instead). *Start* enables printing and requests *lpd* to start printing jobs.

enable and **disable**

Enable and *disable* allow spooling in the local queue to be turned on/off. This will allow/prevent *lpr* from putting new jobs in the spool queue. It is frequently convenient to turn spooling off while testing new line printer filters since the *root* user can still use *lpr* to put jobs in the queue but no one else can. The other main use is to prevent users from putting jobs in the queue when the printer is expected to be unavailable for a long time.

restart

Restart allows ordinary users to restart printer daemons when *lpq* reports that there is no daemon present.

stop

Stop is used to halt a spooling daemon after the current job completes; this also disables printing. This is a clean way to shutdown a printer in order to perform maintenence, etc. Note that users can still enter jobs in a spool queue while a printer is *stopped*.

topq

Topq places jobs at the top of a printer queue. This can be used to reorder high priority jobs since *lpr* only only provides first-come-first-serve ordering of jobs.

7. Troubleshooting

There are a number of messages which may be generated by the the line printer system. This section categorizes the most common and explains the cause for their generation. Where the message indicates a failure, directions are given to remedy the problem.

In the examples below, the name *printer* is the name of the printer. This would be one of the names from the *printcap* database.

7.1. LPR

lpr: *printer*: unknown printer

The *printer* was not found in the *printcap* database. Usually this is a typing mistake; however, it may indicate a missing or incorrect entry in the /etc/printcap file.

lpr: *printer*: jobs queued, but cannot start daemon.

The connection to *lpd* on the local machine failed. This usually means the printer server started at boot time has died or is hung. Check the local socket /dev/printer to be sure it still exists (if it does not exist, there is no *lpd* process running). Use

% ps ax | fgrep lpd

to get a list of process identifiers of running lpd's. The *lpd* to kill is the one which is not listed in any of the "lock" files (the lock file is contained in the spool directory of each printer). Kill the master daemon using the following command.

% kill *pid*

Then remove /dev/printer and restart the daemon (and printer) with the following commands.

% rm /dev/printer
% /usr/lib/lpd

Another possibility is that the *lpr* program is not setuid *root*, setgid *spooling*. This can be checked with

% ls -lg /usr/ucb/lpr

lpr: *printer*: printer queue is disabled

This means the queue was turned off with

% lpc disable *printer*

to prevent *lpr* from putting files in the queue. This is normally done by the system manager when a printer is going to be down for a long time. The printer can be turned back on by a super-user with *lpc*.

7.2. LPQ

waiting for *printer* to become ready (offline ?)

The printer device could not be opened by the daemon. This can happen for a number of reasons, the most common being that the printer is turned off-line. This message can also be generated if the printer is out of paper, the paper is jammed, etc. The actual reason is dependent on the meaning of error codes returned by system device driver. Not all printers supply sufficient information to distinguish when a printer is off-line or having trouble (e.g. a printer connected through a serial line). Another possible cause of this message is some other process, such as an output filter, has an exclusive open on the device. Your only recourse here is to kill off the offending program(s) and restart the printer with *lpc*.

***printer* is ready and printing**

The *lpq* program checks to see if a daemon process exists for *printer* and prints the file *status*. If the daemon is hung, a super user can use *lpc* to abort the current daemon and start a new one.

waiting for *host* to come up

This indicates there is a daemon trying to connect to the remote machine named *host* in order to send the files in the local queue. If the remote machine is up, *lpd* on the remote machine is probably dead or hung and should be restarted as mentioned for *lpr*.

sending to *host*

The files should be in the process of being transferred to the remote *host*. If not, the local daemon should be aborted and started with *lpc*.

Warning: *printer* is down

The printer has been marked as being unavailable with *lpc*.

Warning: no daemon present

The *lpd* process overseeing the spooling queue, as indicated in the "lock" file in that directory, does not exist. This normally occurs only when the daemon has unexpectedly died. The error log file for the printer should be checked for a diagnostic from the deceased process. To restart an *lpd*, use

% lpc restart *printer*

7.3. LPRM

lprm: *printer*: cannot restart printer daemon

This case is the same as when *lpr* prints that the daemon cannot be started.

7.4. LPD

The *lpd* program can write many different messages to the error log file (the file specified in the **lf** entry in *printcap*). Most of these messages are about files which can not be opened and usually indicate the *printcap* file or the protection modes of the files are not correct. Files may also be inaccessible if people manually manipulate the line printer system (i.e. they bypass the *lpr* program).

In addition to messages generated by *lpd*, any of the filters that *lpd* spawns may also log messages to this file.

7.5. LPC

could't start printer

This case is the same as when *lpr* reports that the daemon cannot be started.

cannot examine spool directory

Error messages beginning with "cannot ..." are usually due to incorrect ownership and/or protection mode of the lock file, spooling directory or the *lpc* program.

NAME

printcap – printer capability data base

SYNOPSIS

/etc/printcap

DESCRIPTION

Printcap is a simplified version of the *termcap*(5) data base used to describe line printers. The spooling system accesses the *printcap* file every time it is used, allowing dynamic addition and deletion of printers. Each entry in the data base is used to describe one printer. This data base may not be substituted for, as is possible for *termcap*, because it may allow accounting to be bypassed.

The default printer is normally *lp*, though the environment variable PRINTER may be used to override this. Each spooling utility supports an option, **–P***printer*, to allow explicit naming of a destination printer.

Refer to the *4.2BSD Line Printer Spooler Manual* for a complete discussion on how setup the database for a given printer.

CAPABILITIES

Refer to *termcap* for a description of the file layout.

Name	Type	Default	Description
af	str	NULL	name of accounting file
br	num	none	if lp is a tty, set the baud rate (ioctl call)
cf	str	NULL	cifplot data filter
df	str	NULL	tex data filter (DVI format)
fc	num	0	if lp is a tty, clear flag bits (sgtty.h)
ff	str	"\f"	string to send for a form feed
fo	bool	false	print a form feed when device is opened
fs	num	0	like 'fc' but set bits
gf	str	NULL	graph data filter (plot (3X) format)
ic	bool	false	driver supports (non standard) ioctl to indent printout
if	str	NULL	name of text filter which does accounting
lf	str	"/dev/console"	error logging file name
lo	str	"lock"	name of lock file
lp	str	"/dev/lp"	device name to open for output
mx	num	1000	maximum file size (in BUFSIZ blocks), zero = unlimited
nd	str	NULL	next directory for list of queues (unimplemented)
nf	str	NULL	ditroff data filter (device independent troff)
of	str	NULL	name of output filtering program
pl	num	66	page length (in lines)
pw	num	132	page width (in characters)
px	num	0	page width in pixels (horizontal)
py	num	0	page length in pixels (vertical)
rf	str	NULL	filter for printing FORTRAN style text files
rm	str	NULL	machine name for remote printer
rp	str	"lp"	remote printer name argument
rs	bool	false	restrict remote users to those with local accounts
rw	bool	false	open the printer device for reading and writing
sb	bool	false	short banner (one line only)
sc	bool	false	suppress multiple copies
sd	str	"/usr/spool/lpd"	spool directory

sf	bool	false	suppress form feeds
sh	bool	false	suppress printing of burst page header
st	str	"status"	status file name
tf	str	NULL	troff data filter (cat phototypesetter)
tr	str	NULL	trailer string to print when queue empties
vf	str	NULL	raster image filter
xc	num	0	if lp is a tty, clear local mode bits (tty (4))
xs	num	0	like 'xc' but set bits

Error messages sent to the console have a carriage return and a line feed appended to them, rather than just a line feed.

If the local line printer driver supports indentation, the daemon must understand how to invoke it.

SEE ALSO

termcap(5), lpc(8), lpd(8), pac(8), lpr(1), lpq(1), lprm(1)
4.2BSD Line Printer Spooler Manual

A Fast File System for UNIX*

Revised July 27, 1983

Marshall Kirk McKusick, William N. Joy†,
Samuel J. Leffler‡, Robert S. Fabry

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

ABSTRACT

A reimplementation of the UNIX file system is described. The reimplementation provides substantially higher throughput rates by using more flexible allocation policies, that allow better locality of reference and that can be adapted to a wide range of peripheral and processor characteristics. The new file system clusters data that is sequentially accessed and provides two block sizes to allow fast access for large files while not wasting large amounts of space for small files. File access rates of up to ten times faster than the traditional UNIX file system are experienced. Long needed enhancements to the user interface are discussed. These include a mechanism to lock files, extensions of the name space across file systems, the ability to use arbitrary length file names, and provisions for efficient administrative control of resource usage.

---

* UNIX is a trademark of Bell Laboratories.

†William N. Joy is currently employed by: Sun Microsystems, Inc, 2550 Garcia Avenue, Mountain View, CA 94043

‡Samuel J. Leffler is currently employed by: Lucasfilm Ltd., PO Box 2009, San Rafael, CA 94912

This work was done under grants from the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

TABLE OF CONTENTS

1. Introduction

This paper describes the changes from the original 512 byte UNIX file system to the new one released with the 4.2 Berkeley Software Distribution. It presents the motivations for the changes, the methods used to affect these changes, the rationale behind the design decisions, and a description of the new implementation. This discussion is followed by a summary of the results that have been obtained, directions for future work, and the additions and changes that have been made to the user visible facilities. The paper concludes with a history of the software engineering of the project.

The original UNIX system that runs on the PDP-11† has simple and elegant file system facilities. File system input/output is buffered by the kernel; there are no alignment constraints on data transfers and all operations are made to appear synchronous. All transfers to the disk are in 512 byte blocks, which can be placed arbitrarily within the data area of the file system. No constraints other than available disk space are placed on file growth [Ritchie74], [Thompson79].

When used on the VAX-11 together with other UNIX enhancements, the original 512 byte UNIX file system is incapable of providing the data throughput rates that many applications require. For example, applications that need to do a small amount of processing on a large quantities of data such as VLSI design and image processing, need to have a high throughput from the file system. High throughput rates are also needed by programs with large address spaces that are constructed by mapping files from the file system into virtual memory. Paging data in and out of the file system is likely to occur frequently. This requires a file system providing higher bandwidth than the original 512 byte UNIX one which provides only about two percent of the maximum disk bandwidth or about 20 kilobytes per second per arm [White80], [Smith81b].

Modifications have been made to the UNIX file system to improve its performance. Since the UNIX file system interface is well understood and not inherently slow, this development retained the abstraction and simply changed the underlying implementation to increase its throughput. Consequently users of the system have not been faced with massive software conversion.

Problems with file system performance have been dealt with extensively in the literature; see [Smith81a] for a survey. The UNIX operating system drew many of its ideas from Multics, a large, high performance operating system [Feiertag71]. Other work includes Hydra [Almes78], Spice [Thompson80], and a file system for a lisp environment [Symbolics81a].

A major goal of this project has been to build a file system that is extensible into a networked environment [Holler73]. Other work on network file systems describe centralized file servers [Accetta80], distributed file servers [Dion80], [Luniewski77], [Porcar82], and protocols to reduce the amount of information that must be transferred across a network [Symbolics81b], [Sturgis80].

† DEC, PDP, VAX, MASSBUS, and UNIBUS are trademarks of Digital Equipment Corporation.

2. Old File System

In the old file system developed at Bell Laboratories each disk drive contains one or more file systems.† A file system is described by its super-block, which contains the basic parameters of the file system. These include the number of data blocks in the file system, a count of the maximum number of files, and a pointer to a list of free blocks. All the free blocks in the system are chained together in a linked list. Within the file system are files. Certain files are distinguished as directories and contain pointers to files that may themselves be directories. Every file has a descriptor associated with it called an *inode*. The inode contains information describing ownership of the file, time stamps marking last modification and access times for the file, and an array of indices that point to the data blocks for the file. For the purposes of this section, we assume that the first 8 blocks of the file are directly referenced by values stored in the inode structure itself*. The inode structure may also contain references to indirect blocks containing further data block indices. In a file system with a 512 byte block size, a singly indirect block contains 128 further block addresses, a doubly indirect block contains 128 addresses of further single indirect blocks, and a triply indirect block contains 128 addresses of further doubly indirect blocks.

A traditional 150 megabyte UNIX file system consists of 4 megabytes of inodes followed by 146 megabytes of data. This organization segregates the inode information from the data; thus accessing a file normally incurs a long seek from its inode to its data. Files in a single directory are not typically allocated slots in consecutive locations in the 4 megabytes of inodes, causing many non-consecutive blocks to be accessed when executing operations on all the files in a directory.

The allocation of data blocks to files is also suboptimum. The traditional file system never transfers more than 512 bytes per disk transaction and often finds that the next sequential data block is not on the same cylinder, forcing seeks between 512 byte transfers. The combination of the small block size, limited read-ahead in the system, and many seeks severely limits file system throughput.

The first work at Berkeley on the UNIX file system attempted to improve both reliability and throughput. The reliability was improved by changing the file system so that all modifications of critical information were staged so that they could either be completed or repaired cleanly by a program after a crash [Kowalski78]. The file system performance was improved by a factor of more than two by changing the basic block size from 512 to 1024 bytes. The increase was because of two factors; each disk transfer accessed twice as much data, and most files could be described without need to access through any indirect blocks since the direct blocks contained twice as much data. The file system with these changes will henceforth be referred to as the *old file system.*

This performance improvement gave a strong indication that increasing the block size was a good method for improving throughput. Although the throughput had doubled, the old file system was still using only about four percent of the disk bandwidth. The main problem was that although the free list was initially ordered for optimal access, it quickly became scrambled as files were created and removed. Eventually the free list became entirely random causing files to have their blocks allocated randomly over the disk. This forced the disk to seek before every block access. Although old file systems provided transfer rates of up to 175 kilobytes per second when they were first created, this rate deteriorated to 30 kilobytes per second after a few weeks of moderate use because of randomization of their free block list. There was no way of restoring the performance an old file system except to dump, rebuild, and restore the file system. Another possibility would be to have a process that periodically reorganized the data on the disk to restore locality as suggested by [Maruyama76].

† A file system always resides on a single drive.

* The actual number may vary from system to system, but is usually in the range 5-13.

3. New file system organization

As in the old file system organization each disk drive contains one or more file systems. A file system is described by its super-block, that is located at the beginning of its disk partition. Because the super-block contains critical data it is replicated to protect against catastrophic loss. This is done at the time that the file system is created; since the super-block data does not change, the copies need not be referenced unless a head crash or other hard disk error causes the default super-block to be unusable.

To insure that it is possible to create files as large as 2†32 bytes with only two levels of indirection, the minimum size of a file system block is 4096 bytes. The size of file system blocks can be any power of two greater than or equal to 4096. The block size of the file system is maintained in the super-block so it is possible for file systems with different block sizes to be accessible simultaneously on the same system. The block size must be decided at the time that the file system is created; it cannot be subsequently changed without rebuilding the file system.

The new file system organization partitions the disk into one or more areas called *cylinder groups*. A cylinder group is comprised of one or more consecutive cylinders on a disk. Associated with each cylinder group is some bookkeeping information that includes a redundant copy of the super-block, space for inodes, a bit map describing available blocks in the cylinder group, and summary information describing the usage of data blocks within the cylinder group. For each cylinder group a static number of inodes is allocated at file system creation time. The current policy is to allocate one inode for each 2048 bytes of disk space, expecting this to be far more than will ever be needed.

All the cylinder group bookkeeping information could be placed at the beginning of each cylinder group. However if this approach were used, all the redundant information would be on the top platter. Thus a single hardware failure that destroyed the top platter could cause the loss of all copies of the redundant super-blocks. Thus the cylinder group bookkeeping information begins at a floating offset from the beginning of the cylinder group. The offset for each successive cylinder group is calculated to be about one track further from the beginning of the cylinder group. In this way the redundant information spirals down into the pack so that any single track, cylinder, or platter can be lost without losing all copies of the super-blocks. Except for the first cylinder group, the space between the beginning of the cylinder group and the beginning of the cylinder group information is used for data blocks.†

3.1. Optimizing storage utilization

Data is laid out so that larger blocks can be transferred in a single disk transfer, greatly increasing file system throughput. As an example, consider a file in the new file system composed of 4096 byte data blocks. In the old file system this file would be composed of 1024 byte blocks. By increasing the block size, disk accesses in the new file system may transfer up to four times as much information per disk transaction. In large files, several 4096 byte blocks may be allocated from the same cylinder so that even larger data transfers are possible before initiating a seek.

The main problem with bigger blocks is that most UNIX file systems are composed of many small files. A uniformly large block size wastes space. Table 1 shows the effect of file system block size on the amount of wasted space in the file system. The machine measured to obtain these figures is one of our time sharing systems that has roughly 1.2 Gigabyte of on-line storage. The measurements are based on the active user file systems containing about 920 megabytes of formated space. The space wasted is measured as the percentage of space on the disk not containing user data. As the block size on the disk increases, the waste rises quickly, to an intolerable 45.6% waste with 4096 byte file system blocks.

† While it appears that the first cylinder group could be laid out with its super-block at the "known" location, this would not work for file systems with blocks sizes of 16K or greater, because of the requirement that the cylinder group information must begin at a block boundary.

Space used	% waste	Organization
775.2 Mb	0.0	Data only, no separation between files
807.8 Mb	4.2	Data only, each file starts on 512 byte boundary
828.7 Mb	6.9	512 byte block UNIX file system
866.5 Mb	11.8	1024 byte block UNIX file system
948.5 Mb	22.4	2048 byte block UNIX file system
1128.3 Mb	45.6	4096 byte block UNIX file system

Table 1 – Amount of wasted space as a function of block size.

To be able to use large blocks without undue waste, small files must be stored in a more efficient way. The new file system accomplishes this goal by allowing the division of a single file system block into one or more *fragments*. The file system fragment size is specified at the time that the file system is created; each file system block can be optionally broken into 2, 4, or 8 fragments, each of which is addressable. The lower bound on the size of these fragments is constrained by the disk sector size, typically 512 bytes. The block map associated with each cylinder group records the space availability at the fragment level; to determine block availability, aligned fragments are examined. Figure 1 shows a piece of a map from a 4096/1024 file system.

Bits in map	XXXX	XXOO	OOXX	OOOO
Fragment numbers	0-3	4-7	8-11	12-15
Block numbers	0	1	2	3

Figure 1 – Example layout of blocks and fragments in a 4096/1024 file system.

Each bit in the map records the status of a fragment; an "X" shows that the fragment is in use, while a "O" shows that the fragment is available for allocation. In this example, fragments 0–5, 10, and 11 are in use, while fragments 6–9, and 12–15 are free. Fragments of adjoining blocks cannot be used as a block, even if they are large enough. In this example, fragments 6–9 cannot be coalesced into a block; only fragments 12–15 are available for allocation as a block.

On a file system with a block size of 4096 bytes and a fragment size of 1024 bytes, a file is represented by zero or more 4096 byte blocks of data, and possibly a single fragmented block. If a file system block must be fragmented to obtain space for a small amount of data, the remainder of the block is made available for allocation to other files. As an example consider an 11000 byte file stored on a 4096/1024 byte file system. This file would uses two full size blocks and a 3072 byte fragment. If no 3072 byte fragments are available at the time the file is created, a full size block is split yielding the necessary 3072 byte fragment and an unused 1024 byte fragment. This remaining fragment can be allocated to another file as needed.

The granularity of allocation is the *write* system call. Each time data is written to a file, the system checks to see if the size of the file has increased*. If the file needs to hold the new data, one of three conditions exists:

1) There is enough space left in an already allocated block to hold the new data. The new data is written into the available space in the block.

2) Nothing has been allocated. If the new data contains more than 4096 bytes, a 4096 byte block is allocated and the first 4096 bytes of new data is written there. This process is repeated until less than 4096 bytes of new data remain. If the remaining new data to be written will fit in three or fewer 1024 byte pieces, an unallocated fragment is located, otherwise a 4096 byte block is located. The new data is written into the located piece.

3) A fragment has been allocated. If the number of bytes in the new data plus the number of bytes already in the fragment exceeds 4096 bytes, a 4096 byte block is allocated. The

* A program may be overwriting data in the middle of an existing file in which case space will already be allocated.

contents of the fragment is copied to the beginning of the block and the remainder of the block is filled with the new data. The process then continues as in (2) above. If the number of bytes in the new data plus the number of bytes already in the fragment will fit in three or fewer 1024 byte pieces, an unallocated fragment is located, otherwise a 4096 byte block is located. The contents of the previous fragment appended with the new data is written into the allocated piece.

The problem with allowing only a single fragment on a 4096/1024 byte file system is that data may be potentially copied up to three times as its requirements grow from a 1024 byte fragment to a 2048 byte fragment, then a 3072 byte fragment, and finally a 4096 byte block. The fragment reallocation can be avoided if the user program writes a full block at a time, except for a partial block at the end of the file. Because file systems with different block sizes may coexist on the same system, the file system interface been extended to provide the ability to determine the optimal size for a read or write. For files the optimal size is the block size of the file system on which the file is being accessed. For other objects, such as pipes and sockets, the optimal size is the underlying buffer size. This feature is used by the Standard Input/Output Library, a package used by most user programs. This feature is also used by certain system utilities such as archivers and loaders that do their own input and output management and need the highest possible file system bandwidth.

The space overhead in the 4096/1024 byte new file system organization is empirically observed to be about the same as in the 1024 byte old file system organization. A file system with 4096 byte blocks and 512 byte fragments has about the same amount of space overhead as the 512 byte block UNIX file system. The new file system is more space efficient than the 512 byte or 1024 byte file systems in that it uses the same amount of space for small files while requiring less indexing information for large files. This savings is offset by the need to use more space for keeping track of available free blocks. The net result is about the same disk utilization when the new file systems fragment size equals the old file systems block size.

In order for the layout policies to be effective, the disk cannot be kept completely full. Each file system maintains a parameter that gives the minimum acceptable percentage of file system blocks that can be free. If the the number of free blocks drops below this level only the system administrator can continue to allocate blocks. The value of this parameter can be changed at any time, even when the file system is mounted and active. The transfer rates to be given in section 4 were measured on file systems kept less than 90% full. If the reserve of free blocks is set to zero, the file system throughput rate tends to be cut in half, because of the inability of the file system to localize the blocks in a file. If the performance is impaired because of overfilling, it may be restored by removing enough files to obtain 10% free space. Access speed for files created during periods of little free space can be restored by recreating them once enough space is available. The amount of free space maintained must be added to the percentage of waste when comparing the organizations given in Table 1. Thus, a site running the old 1024 byte UNIX file system wastes 11.8% of the space and one could expect to fit the same amount of data into a 4096/512 byte new file system with 5% free space, since a 512 byte old file system wasted 6.9% of the space.

3.2. File system parameterization

Except for the initial creation of the free list, the old file system ignores the parameters of the underlying hardware. It has no information about either the physical characteristics of the mass storage device, or the hardware that interacts with it. A goal of the new file system is to parameterize the processor capabilities and mass storage characteristics so that blocks can be allocated in an optimum configuration dependent way. Parameters used include the speed of the processor, the hardware support for mass storage transfers, and the characteristics of the mass storage devices. Disk technology is constantly improving and a given installation can have several different disk technologies running on a single processor. Each file system is parameterized so that it can adapt to the characteristics of the disk on which it is placed.

For mass storage devices such as disks, the new file system tries to allocate new blocks on the same cylinder as the previous block in the same file. Optimally, these new blocks will also be well

positioned rotationally. The distance between "rotationally optimal" blocks varies greatly; it can be a consecutive block or a rotationally delayed block depending on system characteristics. On a processor with a channel that does not require any processor intervention between mass storage transfer requests, two consecutive disk blocks often can be accessed without suffering lost time because of an intervening disk revolution. For processors without such channels, the main processor must field an interrupt and prepare for a new disk transfer. The expected time to service this interrupt and schedule a new disk transfer depends on the speed of the main processor.

The physical characteristics of each disk include the number of blocks per track and the rate at which the disk spins. The allocation policy routines use this information to calculate the number of milliseconds required to skip over a block. The characteristics of the processor include the expected time to schedule an interrupt. Given the previous block allocated to a file, the allocation routines calculate the number of blocks to skip over so that the next block in a file will be coming into position under the disk head in the expected amount of time that it takes to start a new disk transfer operation. For programs that sequentially access large amounts of data, this strategy minimizes the amount of time spent waiting for the disk to position itself.

To ease the calculation of finding rotationally optimal blocks, the cylinder group summary information includes a count of the availability of blocks at different rotational positions. Eight rotational positions are distinguished, so the resolution of the summary information is 2 milliseconds for a typical 3600 revolution per minute drive.

The parameter that defines the minimum number of milliseconds between the completion of a data transfer and the initiation of another data transfer on the same cylinder can be changed at any time, even when the file system is mounted and active. If a file system is parameterized to lay out blocks with rotational separation of 2 milliseconds, and the disk pack is then moved to a system that has a processor requiring 4 milliseconds to schedule a disk operation, the throughput will drop precipitously because of lost disk revolutions on nearly every block. If the eventual target machine is known, the file system can be parameterized for it even though it is initially created on a different processor. Even if the move is not known in advance, the rotational layout delay can be reconfigured after the disk is moved so that all further allocation is done based on the characteristics of the new host.

3.3. Layout policies

The file system policies are divided into two distinct parts. At the top level are global policies that use file system wide summary information to make decisions regarding the placement of new inodes and data blocks. These routines are responsible for deciding the placement of new directories and files. They also calculate rotationally optimal block layouts, and decide when to force a long seek to a new cylinder group because there are insufficient blocks left in the current cylinder group to do reasonable layouts. Below the global policy routines are the local allocation routines that use a locally optimal scheme to lay out data blocks.

Two methods for improving file system performance are to increase the locality of reference to minimize seek latency as described by [Trivedi80], and to improve the layout of data to make larger transfers possible as described by [Nevalainen77]. The global layout policies try to improve performance by clustering related information. They cannot attempt to localize all data references, but must also try to spread unrelated data among different cylinder groups. If too much localization is attempted, the local cylinder group may run out of space forcing the data to be scattered to non-local cylinder groups. Taken to an extreme, total localization can result in a single huge cluster of data resembling the old file system. The global policies try to balance the two conflicting goals of localizing data that is concurrently accessed while spreading out unrelated data.

One allocatable resource is inodes. Inodes are used to describe both files and directories. Files in a directory are frequently accessed together. For example the "list directory" command often accesses the inode for each file in a directory. The layout policy tries to place all the files in a directory in the same cylinder group. To ensure that files are allocated throughout the disk, a different policy is used for directory allocation. A new directory is placed in the cylinder group that has a greater than average number of free inodes, and the fewest number of directories in it

already. The intent of this policy is to allow the file clustering policy to succeed most of the time. The allocation of inodes within a cylinder group is done using a next free strategy. Although this allocates the inodes randomly within a cylinder group, all the inodes for each cylinder group can be read with 4 to 8 disk transfers. This puts a small and constant upper bound on the number of disk transfers required to access all the inodes for all the files in a directory as compared to the old file system where typically, one disk transfer is needed to get the inode for each file in a directory.

The other major resource is the data blocks. Since data blocks for a file are typically accessed together, the policy routines try to place all the data blocks for a file in the same cylinder group, preferably rotationally optimally on the same cylinder. The problem with allocating all the data blocks in the same cylinder group is that large files will quickly use up available space in the cylinder group, forcing a spill over to other areas. Using up all the space in a cylinder group has the added drawback that future allocations for any file in the cylinder group will also spill to other areas. Ideally none of the cylinder groups should ever become completely full. The solution devised is to redirect block allocation to a newly chosen cylinder group when a file exceeds 32 kilobytes, and at every megabyte thereafter. The newly chosen cylinder group is selected from those cylinder groups that have a greater than average number of free blocks left. Although big files tend to be spread out over the disk, a megabyte of data is typically accessible before a long seek must be performed, and the cost of one long seek per megabyte is small.

The global policy routines call local allocation routines with requests for specific blocks. The local allocation routines will always allocate the requested block if it is free. If the requested block is not available, the allocator allocates a free block of the requested size that is rotationally closest to the requested block. If the global layout policies had complete information, they could always request unused blocks and the allocation routines would be reduced to simple bookkeeping. However, maintaining complete information is costly; thus the implementation of the global layout policy uses heuristic guesses based on partial information.

If a requested block is not available the local allocator uses a four level allocation strategy:

1) Use the available block rotationally closest to the requested block on the same cylinder.
2) If there are no blocks available on the same cylinder, use a block within the same cylinder group.
3) If the cylinder group is entirely full, quadratically rehash among the cylinder groups looking for a free block.
4) Finally if the rehash fails, apply an exhaustive search.

The use of quadratic rehash is prompted by studies of symbol table strategies used in programming languages. File systems that are parameterized to maintain at least 10% free space almost never use this strategy; file systems that are run without maintaining any free space typically have so few free blocks that almost any allocation is random. Consequently the most important characteristic of the strategy used when the file system is low on space is that it be fast.

4. Performance

Ultimately, the proof of the effectiveness of the algorithms described in the previous section is the long term performance of the new file system.

Our empiric studies have shown that the inode layout policy has been effective. When running the "list directory" command on a large directory that itself contains many directories, the number of disk accesses for inodes is cut by a factor of two. The improvements are even more dramatic for large directories containing only files, disk accesses for inodes being cut by a factor of eight. This is most encouraging for programs such as spooling daemons that access many small files, since these programs tend to flood the disk request queue on the old file system.

Table 2 summarizes the measured throughput of the new file system. Several comments need to be made about the conditions under which these tests were run. The test programs measure the rate that user programs can transfer data to or from a file without performing any processing on it. These programs must write enough data to insure that buffering in the operating system does not affect the results. They should also be run at least three times in succession; the first to get the system into a known state and the second two to insure that the experiment has stabilized and is repeatable. The methodology and test results are discussed in detail in [Kridle83]†. The systems were running multi-user but were otherwise quiescent. There was no contention for either the cpu or the disk arm. The only difference between the UNIBUS and MASSBUS tests was the controller. All tests used an Ampex Capricorn 330 Megabyte Winchester disk. As Table 2 shows, all file system test runs were on a VAX 11/750. All file systems had been in production use for at least a month before being measured.

Type of File System	Processor and Bus Measured	Read Speed	Read Bandwidth	Read % CPU
old 1024	750/UNIBUS	29 Kbytes/sec	29/1100 3%	11%
new 4096/1024	750/UNIBUS	221 Kbytes/sec	221/1100 20%	43%
new 8192/1024	750/UNIBUS	233 Kbytes/sec	233/1100 21%	29%
new 4096/1024	750/MASSBUS	466 Kbytes/sec	466/1200 39%	73%
new 8192/1024	750/MASSBUS	466 Kbytes/sec	466/1200 39%	54%

Table 2a – Reading rates of the old and new UNIX file systems.

Type of File System	Processor and Bus Measured	Write Speed	Write Bandwidth	Write % CPU
old 1024	750/UNIBUS	48 Kbytes/sec	48/1100 4%	29%
new 4096/1024	750/UNIBUS	142 Kbytes/sec	142/1100 13%	43%
new 8192/1024	750/UNIBUS	215 Kbytes/sec	215/1100 19%	46%
new 4096/1024	750/MASSBUS	323 Kbytes/sec	323/1200 27%	94%
new 8192/1024	750/MASSBUS	466 Kbytes/sec	466/1200 39%	95%

Table 2b – Writing rates of the old and new UNIX file systems.

Unlike the old file system, the transfer rates for the new file system do not appear to change over time. The throughput rate is tied much more strongly to the amount of free space that is maintained. The measurements in Table 2 were based on a file system run with 10% free space. Synthetic work loads suggest the performance deteriorates to about half the throughput rates given in Table 2 when no free space is maintained.

The percentage of bandwidth given in Table 2 is a measure of the effective utilization of the disk by the file system. An upper bound on the transfer rate from the disk is measured by doing 65536* byte reads from contiguous tracks on the disk. The bandwidth is calculated by comparing

† A UNIX command that is similar to the reading test that we used is, "cp file /dev/null", where "file" is eight Megabytes long.

* This number, 65536, is the maximal I/O size supported by the VAX hardware; it is a remnant of the system's PDP-11 ancestry.

the data rates the file system is able to achieve as a percentage of this rate. Using this metric, the old file system is only able to use about 3-4% of the disk bandwidth, while the new file system uses up to 39% of the bandwidth.

In the new file system, the reading rate is always at least as fast as the writing rate. This is to be expected since the kernel must do more work when allocating blocks than when simply reading them. Note that the write rates are about the same as the read rates in the 8192 byte block file system; the write rates are slower than the read rates in the 4096 byte block file system. The slower write rates occur because the kernel has to do twice as many disk allocations per second, and the processor is unable to keep up with the disk transfer rate.

In contrast the old file system is about 50% faster at writing files than reading them. This is because the *write* system call is asynchronous and the kernel can generate disk transfer requests much faster than they can be serviced, hence disk transfers build up in the disk buffer cache. Because the disk buffer cache is sorted by minimum seek order, the average seek between the scheduled disk writes is much less than they would be if the data blocks are written out in the order in which they are generated. However when the file is read, the *read* system call is processed synchronously so the disk blocks must be retrieved from the disk in the order in which they are allocated. This forces the disk scheduler to do long seeks resulting in a lower throughput rate.

The performance of the new file system is currently limited by a memory to memory copy operation because it transfers data from the disk into buffers in the kernel address space and then spends 40% of the processor cycles copying these buffers to user address space. If the buffers in both address spaces are properly aligned, this transfer can be affected without copying by using the VAX virtual memory management hardware. This is especially desirable when large amounts of data are to be transferred. We did not implement this because it would change the semantics of the file system in two major ways; user programs would be required to allocate buffers on page boundaries, and data would disappear from buffers after being written.

Greater disk throughput could be achieved by rewriting the disk drivers to chain together kernel buffers. This would allow files to be allocated to contiguous disk blocks that could be read in a single disk transaction. Most disks contain either 32 or 48 512 byte sectors per track. The inability to use contiguous disk blocks effectively limits the performance on these disks to less than fifty percent of the available bandwidth. Since each track has a multiple of sixteen sectors it holds exactly two or three 8192 byte file system blocks, or four or six 4096 byte file system blocks. If the the next block for a file cannot be laid out contiguously, then the minimum spacing to the next allocatable block on any platter is between a sixth and a half a revolution. The implication of this is that the best possible layout without contiguous blocks uses only half of the bandwidth of any given track. If each track contains an odd number of sectors, then it is possible to resolve the rotational delay to any number of sectors by finding a block that begins at the desired rotational position on another track. The reason that block chaining has not been implemented is because it would require rewriting all the disk drivers in the system, and the current throughput rates are already limited by the speed of the available processors.

Currently only one block is allocated to a file at a time. A technique used by the DEMOS file system when it finds that a file is growing rapidly, is to preallocate several blocks at once, releasing them when the file is closed if they remain unused. By batching up the allocation the system can reduce the overhead of allocating at each write, and it can cut down on the number of disk writes needed to keep the block pointers on the disk synchronized with the block allocation [Powell79].

5. File system functional enhancements

The speed enhancements to the UNIX file system did not require any changes to the semantics or data structures viewed by the users. However several changes have been generally desired for some time but have not been introduced because they would require users to dump and restore all their file systems. Since the new file system already requires that all existing file systems be dumped and restored, these functional enhancements have been introduced at this time.

5.1. Long file names

File names can now be of nearly arbitrary length. The only user programs affected by this change are those that access directories. To maintain portability among UNIX systems that are not running the new file system, a set of directory access routines have been introduced that provide a uniform interface to directories on both old and new systems.

Directories are allocated in units of 512 bytes. This size is chosen so that each allocation can be transferred to disk in a single atomic operation. Each allocation unit contains variable-length directory entries. Each entry is wholly contained in a single allocation unit. The first three fields of a directory entry are fixed and contain an inode number, the length of the entry, and the length of the name contained in the entry. Following this fixed size information is the null terminated name, padded to a 4 byte boundary. The maximum length of a name in a directory is currently 255 characters.

Free space in a directory is held by entries that have a record length that exceeds the space required by the directory entry itself. All the bytes in a directory unit are claimed by the directory entries. This normally results in the last entry in a directory being large. When entries are deleted from a directory, the space is returned to the previous entry in the same directory unit by increasing its length. If the first entry of a directory unit is free, then its inode number is set to zero to show that it is unallocated.

5.2. File locking

The old file system had no provision for locking files. Processes that needed to synchronize the updates of a file had to create a separate "lock" file to synchronize their updates. A process would try to create a "lock" file. If the creation succeeded, then it could proceed with its update; if the creation failed, then it would wait, and try again. This mechanism had three drawbacks. Processes consumed CPU time, by looping over attempts to create locks. Locks were left lying around following system crashes and had to be cleaned up by hand. Finally, processes running as system administrator are always permitted to create files, so they had to use a different mechanism. While it is possible to get around all these problems, the solutions are not straight-forward, so a mechanism for locking files has been added.

The most general schemes allow processes to concurrently update a file. Several of these techniques are discussed in [Peterson83]. A simpler technique is to simply serialize access with locks. To attain reasonable efficiency, certain applications require the ability to lock pieces of a file. Locking down to the byte level has been implemented in the Onyx file system by [Bass81]. However, for the applications that currently run on the system, a mechanism that locks at the granularity of a file is sufficient.

Locking schemes fall into two classes, those using hard locks and those using advisory locks. The primary difference between advisory locks and hard locks is the decision of when to override them. A hard lock is always enforced whenever a program tries to access a file; an advisory lock is only applied when it is requested by a program. Thus advisory locks are only effective when all programs accessing a file use the locking scheme. With hard locks there must be some override policy implemented in the kernel, with advisory locks the policy is implemented by the user programs. In the UNIX system, programs with system administrator privilege can override any protection scheme. Because many of the programs that need to use locks run as system administrators, we chose to implement advisory locks rather than create a protection scheme that was contrary to the UNIX philosophy or could not be used by system administration programs.

The file locking facilities allow cooperating programs to apply advisory *shared* or *exclusive* locks on files. Only one process has an exclusive lock on a file while multiple shared locks may be present. Both shared and exclusive locks cannot be present on a file at the same time. If any lock is requested when another process holds an exclusive lock, or an exclusive lock is requested when another process holds any lock, the open will block until the lock can be gained. Because shared and exclusive locks are advisory only, even if a process has obtained a lock on a file, another process can override the lock by opening the same file without a lock.

Locks can be applied or removed on open files, so that locks can be manipulated without needing to close and reopen the file. This is useful, for example, when a process wishes to open a file with a shared lock to read some information, to determine whether an update is required. It can then get an exclusive lock so that it can do a read, modify, and write to update the file in a consistent manner.

A request for a lock will cause the process to block if the lock can not be immediately obtained. In certain instances this is unsatisfactory. For example, a process that wants only to check if a lock is present would require a separate mechanism to find out this information. Consequently, a process may specify that its locking request should return with an error if a lock can not be immediately obtained. Being able to poll for a lock is useful to "daemon" processes that wish to service a spooling area. If the first instance of the daemon locks the directory where spooling takes place, later daemon processes can easily check to see if an active daemon exists. Since the lock is removed when the process exits or the system crashes, there is no problem with unintentional locks files that must be cleared by hand.

Almost no deadlock detection is attempted. The only deadlock detection made by the system is that the file descriptor to which a lock is applied does not currently have a lock of the same type (i.e. the second of two successive calls to apply a lock of the same type will fail). Thus a process can deadlock itself by requesting locks on two separate file descriptors for the same object.

5.3. Symbolic links

The 512 byte UNIX file system allows multiple directory entries in the same file system to reference a single file. The link concept is fundamental; files do not live in directories, but exist separately and are referenced by links. When all the links are removed, the file is deallocated. This style of links does not allow references across physical file systems, nor does it support inter-machine linkage. To avoid these limitations *symbolic links* have been added similar to the scheme used by Multics [Feiertag71].

A symbolic link is implemented as a file that contains a pathname. When the system encounters a symbolic link while interpreting a component of a pathname, the contents of the symbolic link is prepended to the rest of the pathname, and this name is interpreted to yield the resulting pathname. If the symbolic link contains an absolute pathname, the absolute pathname is used, otherwise the contents of the symbolic link is evaluated relative to the location of the link in the file hierarchy.

Normally programs do not want to be aware that there is a symbolic link in a pathname that they are using. However certain system utilities must be able to detect and manipulate symbolic links. Three new system calls provide the ability to detect, read, and write symbolic links, and seven system utilities were modified to use these calls.

In future Berkeley software distributions it will be possible to mount file systems from other machines within a local file system. When this occurs, it will be possible to create symbolic links that span machines.

5.4. Rename

Programs that create new versions of data files typically create the new version as a temporary file and then rename the temporary file with the original name of the data file. In the old UNIX file systems the renaming required three calls to the system. If the program were interrupted or the system crashed between these calls, the data file could be left with only its temporary name.

To eliminate this possibility a single system call has been added that performs the rename in an atomic fashion to guarantee the existence of the original name.

In addition, the rename facility allows directories to be moved around in the directory tree hierarchy. The rename system call performs special validation checks to insure that the directory tree structure is not corrupted by the creation of loops or inaccessible directories. Such corruption would occur if a parent directory were moved into one of its descendants. The validation check requires tracing the ancestry of the target directory to insure that it does not include the directory being moved.

5.5. Quotas

The UNIX system has traditionally attempted to share all available resources to the greatest extent possible. Thus any single user can allocate all the available space in the file system. In certain environments this is unacceptable. Consequently, a quota mechanism has been added for restricting the amount of file system resources that a user can obtain. The quota mechanism sets limits on both the number of files and the number of disk blocks that a user may allocate. A separate quota can be set for each user on each file system. Each resource is given both a hard and a soft limit. When a program exceeds a soft limit, a warning is printed on the users terminal; the offending program is not terminated unless it exceeds its hard limit. The idea is that users should stay below their soft limit between login sessions, but they may use more space while they are actively working. To encourage this behavior, users are warned when logging in if they are over any of their soft limits. If they fail to correct the problem for too many login sessions, they are eventually reprimanded by having their soft limit enforced as their hard limit.

6. Software engineering

The preliminary design was done by Bill Joy in late 1980; he presented the design at The USENIX Conference held in San Francisco in January 1981. The implementation of his design was done by Kirk McKusick in the summer of 1981. Most of the new system calls were implemented by Sam Leffler. The code for enforcing quotas was implemented by Robert Elz at the University of Melbourne.

To understand how the project was done it is necessary to understand the interfaces that the UNIX system provides to the hardware mass storage systems. At the lowest level is a *raw disk*. This interface provides access to the disk as a linear array of sectors. Normally this interface is only used by programs that need to do disk to disk copies or that wish to dump file systems. However, user programs with proper access rights can also access this interface. A disk is usually formated with a file system that is interpreted by the UNIX system to provide a directory hierarchy and files. The UNIX system interprets and multiplexes requests from user programs to create, read, write, and delete files by allocating and freeing inodes and data blocks. The interpretation of the data on the disk could be done by the user programs themselves. The reason that it is done by the UNIX system is to synchronize the user requests, so that two processes do not attempt to allocate or modify the same resource simultaneously. It also allows access to be restricted at the file level rather than at the disk level and allows the common file system routines to be shared between processes.

The implementation of the new file system amounted to using a different scheme for formating and interpreting the disk. Since the synchronization and disk access routines themselves were not being changed, the changes to the file system could be developed by moving the file system interpretation routines out of the kernel and into a user program. Thus, the first step was to extract the file system code for the old file system from the UNIX kernel and change its requests to the disk driver to accesses to a raw disk. This produced a library of routines that mapped what would normally be system calls into read or write operations on the raw disk. This library was then debugged by linking it into the system utilities that copy, remove, archive, and restore files.

A new cross file system utility was written that copied files from the simulated file system to the one implemented by the kernel. This was accomplished by calling the simulation library to do a read, and then writing the resultant data by using the conventional write system call. A similar utility copied data from the kernel to the simulated file system by doing a conventional read system call and then writing the resultant data using the simulated file system library.

The second step was to rewrite the file system simulation library to interpret the new file system. By linking the new simulation library into the cross file system copying utility, it was possible to easily copy files from the old file system into the new one and from the new one to the old one. Having the file system interpretation implemented in user code had several major benefits. These included being able to use the standard system tools such as the debuggers to set breakpoints and single step through the code. When bugs were discovered, the offending problem could be fixed and tested without the need to reboot the machine. There was never a period where it was necessary to maintain two concurrent file systems in the kernel. Finally it was not necessary to dedicate a machine entirely to file system development, except for a brief period while the new file system was boot strapped.

The final step was to merge the new file system back into the UNIX kernel. This was done in less than two weeks, since the only bugs remaining were those that involved interfacing to the synchronization routines that could not be tested in the simulated system. Again the simulation system proved useful since it enabled files to be easily copied between old and new file systems regardless of which file system was running in the kernel. This greatly reduced the number of times that the system had to be rebooted.

The total design and debug time took about one man year. Most of the work was done on the file system utilities, and changing all the user programs to use the new facilities. The code changes in the kernel were minor, involving the addition of only about 800 lines of code (including comments).

Acknowledgements

We thank Robert Elz for his ongoing interest in the new file system, and for adding disk quotas in a rational and efficient manner. We also acknowledge Dennis Ritchie for his suggestions on the appropriate modifications to the user interface. We appreciate Michael Powell's explanations on how the DEMOS file system worked; many of his ideas were used in this implementation. Special commendation goes to Peter Kessler and Robert Henry for acting like real users during the early debugging stage when files were less stable than they should have been. Finally we thank our sponsors, the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

References

[Accetta80] Accetta, M., Robertson, G., Satyanarayanan, M., and Thompson, M. "The Design of a Network-Based Central File System", Carnegie-Mellon University, Dept of Computer Science Tech Report, #CMU-CS-80-134

[Almes78] Almes, G., and Robertson, G. "An Extensible File System for Hydra" Proceedings of the Third International Conference on Software Engineering, IEEE, May 1978.

[Bass81] Bass, J. "Implementation Description for File Locking", Onyx Systems Inc, 73 E. Trimble Rd, San Jose, CA 95131 Jan 1981.

[Dion80] Dion, J. "The Cambridge File Server", Operating Systems Review, 14, 4. Oct 1980. pp 26-35

[Eswaran74] Eswaran, K. "Placement of records in a file and file allocation in a computer network", Proceedings IFIPS, 1974. pp 304-307

[Holler73] Holler, J. "Files in Computer Networks", First European Workshop on Computer Networks, April 1973. pp 381-396

[Feiertag71] Feiertag, R. J. and Organick, E. I., "The Multics Input-Output System", Proceedings of the Third Symposium on Operating Systems Principles, ACM, Oct 1971. pp 35-41

[Kridle83] Kridle, R., and McKusick, M., "Performance Effects of Disk Subsystem Choices for VAX Systems Running 4.2BSD UNIX", Computer Systems Research Group, Dept of EECS, Berkeley, CA 94720, Technical Report #8.

[Kowalski78] Kowalski, T. "FSCK - The UNIX System Check Program", Bell Laboratory, Murray Hill, NJ 07974. March 1978

[Luniewski77] Luniewski, A. "File Allocation in a Distributed System", MIT Laboratory for Computer Science, Dec 1977.

[Maruyama76] Maruyama, K., and Smith, S. "Optimal reorganization of Distributed Space Disk Files", Communications of the ACM, 19, 11. Nov 1976. pp 634-642

[Nevalainen77] Nevalainen, O., Vesterinen, M. "Determining Blocking Factors for Sequential Files by Heuristic Methods", The Computer Journal, 20, 3. Aug 1977. pp 245-247

[Peterson83] Peterson, G. "Concurrent Reading While Writing", ACM Transactions on Programming Languages and Systems, ACM, 5, 1. Jan 1983. pp 46-55

[Powell79] Powell, M. "The DEMOS File System", Proceedings of the Sixth Symposium on Operating Systems Principles, ACM, Nov 1977. pp 33-42

[Porcar82] Porcar, J. "File Migration in Distributed Computer Systems", Ph.D. Thesis, Lawrence Berkeley Laboratory Tech Report #LBL-14763.

[Ritchie74] Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System", CACM 17, 7. July 1974. pp 365-375

[Smith81a] Smith, A. "Input/Output Optimization and Disk Architectures: A Survey", Performance and Evaluation 1. Jan 1981. pp 104-117

[Smith81b] Smith, A. "Bibliography on File and I/O System Optimization and Related Topics", Operating Systems Review, 15, 4. Oct 1981. pp 39-54

[Sturgis80] Sturgis, H., Mitchell, J., and Israel, J. "Issues in the Design and Use of a Distributed File System", Operating Systems Review, 14, 3. pp 55-79

[Symbolics81a] "Symbolics File System", Symbolics Inc, 9600 DeSoto Ave, Chatsworth, CA 91311 Aug 1981.

[Symbolics81b] "Chaosnet FILE Protocol". Symbolics Inc, 9600 DeSoto Ave, Chatsworth, CA 91311 Sept 1981.

[Thompson79] Thompson, K. "UNIX Implementation", Section 31, Volume 2B, UNIX Programmers Manual, Bell Laboratory, Murray Hill, NJ 07974. Jan 1979

[Thompson80] Thompson, M. "Spice File System", Carnegie-Mellon University, Dept of Computer Science Tech Report, #CMU-CS-80-???

[Trivedi80] Trivedi, K. "Optimal Selection of CPU Speed, Device Capabilities, and File Assignments", Journal of the ACM, 27, 3. July 1980. pp 457-473

[White80] White, R. M. "Disk Storage Technology", Scientific American, 243(2), August 1980.

4.2BSD Networking Implementation Notes

Revised July, 1983

Samuel J. Leffler, William N. Joy, Robert S. Fabry

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

(415) 642-7780

ABSTRACT

This report describes the internal structure of the networking facilities developed for the 4.2BSD version of the UNIX* operating system for the VAX†. These facilities are based on several central abstractions which structure the external (user) view of network communication as well as the internal (system) implementation.

The report documents the internal structure of the networking system. The "4.2BSD System Manual" provides a description of the user interface to the networking facilities.

* UNIX is a trademark of Bell Laboratories.
† DEC, VAX, DECnet, and UNIBUS are trademarks of Digital Equipment Corporation.

TABLE OF CONTENTS

1. Introduction

This report describes the internal structure of facilities added to the 4.2BSD version of the UNIX operating system for the VAX. The system facilities provide a uniform user interface to networking within UNIX. In addition, the implementation introduces a structure for network communications which may be used by system implementors in adding new networking facilities. The internal structure is not visible to the user, rather it is intended to aid implementors of communication protocols and network services by providing a framework which promotes code sharing and minimizes implementation effort.

The reader is expected to be familiar with the C programming language and system interface, as described in the *4.2BSD System Manual* [Joy82a]. Basic understanding of network communication concepts is assumed; where required any additional ideas are introduced.

The remainder of this document provides a description of the system internals, avoiding, when possible, those portions which are utilized only by the interprocess communication facilities.

2. Overview

If we consider the International Standards Organization's (ISO) Open System Interconnection (OSI) model of network communication [ISO81] [Zimmermann80], the networking facilities described here correspond to a portion of the session layer (layer 3) and all of the transport and network layers (layers 2 and 1, respectively).

The network layer provides possibly imperfect data transport services with minimal addressing structure. Addressing at this level is normally host to host, with implicit or explicit routing optionally supported by the communicating agents.

At the transport layer the notions of reliable transfer, data sequencing, flow control, and service addressing are normally included. Reliability is usually managed by explicit acknowledgement of data delivered. Failure to acknowledge a transfer results in retransmission of the data. Sequencing may be handled by tagging each message handed to the network layer by a *sequence number* and maintaining state at the endpoints of communication to utilize received sequence numbers in reordering data which arrives out of order.

The session layer facilities may provide forms of addressing which are mapped into formats required by the transport layer, service authentication and client authentication, etc. Various systems also provide services such as data encryption and address and protocol translation.

The following sections begin by describing some of the common data structures and utility routines, then examine the internal layering. The contents of each layer and its interface are considered. Certain of the interfaces are protocol implementation specific. For these cases examples have been drawn from the Internet [Cerf78] protocol family. Later sections cover routing issues, the design of the raw socket interface and other miscellaneous topics.

3. Goals

The networking system was designed with the goal of supporting multiple *protocol families* and addressing styles. This required information to be "hidden" in common data structures which could be manipulated by all the pieces of the system, but which required interpretation only by the protocols which "controlled" it. The system described here attempts to minimize the use of shared data structures to those kept by a suite of protocols (a *protocol family*), and those used for rendezvous between "synchronous" and "asynchronous" portions of the system (e.g. queues of data packets are filled at interrupt time and emptied based on user requests).

A major goal of the system was to provide a framework within which new protocols and hardware could be easily be supported. To this end, a great deal of effort has been extended to create utility routines which hide many of the more complex and/or hardware dependent chores of networking. Later sections describe the utility routines and the underlying data structures they manipulate.

4. Internal address representation

Common to all portions of the system are two data structures. These structures are used to represent addresses and various data objects. Addresses, internally are described by the *sockaddr* structure,

```
struct sockaddr {
        short       sa_family;          /* data format identifier */
        char        sa_data[14];        /* address */
};
```

All addresses belong to one or more *address families* which define their format and interpretation. The *sa_family* field indicates which address family the address belongs to, the *sa_data* field contains the actual data value. The size of the data field, 14 bytes, was selected based on a study of current address formats*.

* Later versions of the system support variable length addresses.

5. Memory management

A single mechanism is used for data storage: memory buffers, or *mbuf*s. An mbuf is a structure of the form:

```
struct mbuf {
        struct      mbuf *m_next;     /* next buffer in chain */
        u_long      m_off;            /* offset of data */
        short       m_len;            /* amount of data in this mbuf */
        short       m_type;           /* mbuf type (accounting) */
        u_char      m_dat[MLEN];      /* data storage */
        struct      mbuf *m_act;      /* link in higher-level mbuf list */
};
```

The *m_next* field is used to chain mbufs together on linked lists, while the *m_act* field allows lists of mbufs to be accumulated. By convention, the mbufs common to a single object (for example, a packet) are chained together with the *m_next* field, while groups of objects are linked via the *m_act* field (possibly when in a queue).

Each mbuf has a small data area for storing information, *m_dat*. The *m_len* field indicates the amount of data, while the *m_off* field is an offset to the beginning of the data from the base of the mbuf. Thus, for example, the macro *mtod*, which converts a pointer to an mbuf to a pointer to the data stored in the mbuf, has the form

```
#define  mtod(x,t)          ((t)((int)(x) + (x)->m_off))
```

(note the *t* parameter, a C type cast, is used to cast the resultant pointer for proper assignment).

In addition to storing data directly in the mbuf's data area, data of page size may be also be stored in a separate area of memory. The mbuf utility routines maintain a pool of pages for this purpose and manipulate a private page map for such pages. The virtual addresses of these data pages precede those of mbufs, so when pages of data are separated from an mbuf, the mbuf data offset is a negative value. An array of reference counts on pages is also maintained so that copies of pages may be made without core to core copying (copies are created simply by duplicating the relevant page table entries in the data page map and incrementing the associated reference counts for the pages). Separate data pages are currently used only when copying data from a user process into the kernel, and when bringing data in at the hardware level. Routines which manipulate mbufs are not normally aware if data is stored directly in the mbuf data array, or if it is kept in separate pages.

The following utility routines are available for manipulating mbuf chains:

m = m_copy(m0, off, len);
: The *m_copy* routine create a copy of all, or part, of a list of the mbufs in *m0*. *Len* bytes of data, starting *off* bytes from the front of the chain, are copied. Where possible, reference counts on pages are used instead of core to core copies. The original mbuf chain must have at least *off* + *len* bytes of data. If *len* is specified as M_COPYALL, all the data present, offset as before, is copied.

m_cat(m, n);
: The mbuf chain, *n*, is appended to the end of *m*. Where possible, compaction is performed.

m_adj(m, diff);
: The mbuf chain, *m* is adjusted in size by *diff* bytes. If *diff* is non-negative, *diff* bytes are shaved off the front of the mbuf chain. If *diff* is negative, the alteration is performed from back to front. No space is reclaimed in this operation, alterations are accomplished by changing the *m_len* and *m_off* fields of mbufs.

m = m_pullup(m0, size);
: After a successful call to *m_pullup*, the mbuf at the head of the returned list, *m*, is guaranteed to have at least *size* bytes of data in contiguous memory (allowing access via a pointer, obtained using the *mtod* macro). If the original data was less than *size* bytes long,

len was greater than the size of an mbuf data area (112 bytes), or required resources were unavailable, *m* is 0 and the original mbuf chain is deallocated.

This routine is particularly useful when verifying packet header lengths on reception. For example, if a packet is received and only 8 of the necessary 16 bytes required for a valid packet header are present at the head of the list of mbufs representing the packet, the remaining 8 bytes may be "pulled up" with a single *m_pullup* call. If the call fails the invalid packet will have been discarded.

By insuring mbufs always reside on 128 byte boundaries it is possible to always locate the mbuf associated with a data area by masking off the low bits of the virtual address. This allows modules to store data structures in mbufs and pass them around without concern for locating the original mbuf when it comes time to free the structure. The *dtom* macro is used to convert a pointer into an mbuf's data area to a pointer to the mbuf,

```
#define     dtom(x)     ((struct mbuf *)((int)x & ~(MSIZE-1)))
```

Mbufs are used for dynamically allocated data structures such as sockets, as well as memory allocated for packets. Statistics are maintained on mbuf usage and can be viewed by users using the *netstat*(1) program.

6. Internal layering

The internal structure of the network system is divided into three layers. These layers correspond to the services provided by the socket abstraction, those provided by the communication protocols, and those provided by the hardware interfaces. The communication protocols are normally layered into two or more individual cooperating layers, though they are collectively viewed in the system as one layer providing services supportive of the appropriate socket abstraction.

The following sections describe the properties of each layer in the system and the interfaces each must conform to.

6.1. Socket layer

The socket layer deals with the interprocess communications facilities provided by the system. A socket is a bidirectional endpoint of communication which is "typed" by the semantics of communication it supports. The system calls described in the *4.2BSD System Manual* are used to manipulate sockets.

A socket consists of the following data structure:

```
struct socket {
        short       so_type;        /* generic type */
        short       so_options;     /* from socket call */
        short       so_linger;      /* time to linger while closing */
        short       so_state;       /* internal state flags */
        caddr_t     so_pcb;         /* protocol control block */
        struct      protosw *so_proto; /* protocol handle */
        struct      socket *so_head; /* back pointer to accept socket */
        struct      socket *so_q0;  /* queue of partial connections */
        short       so_q0len;       /* partials on so_q0 */
        struct      socket *so_q;   /* queue of incoming connections */
        short       so_qlen;        /* number of connections on so_q */
        short       so_qlimit;      /* max number queued connections */
        struct      sockbuf so_snd; /* send queue */
        struct      sockbuf so_rcv; /* receive queue */
        short       so_timeo;       /* connection timeout */
        u_short     so_error;       /* error affecting connection */
        short       so_oobmark;     /* chars to oob mark */
        short       so_pgrp;        /* pgrp for signals */
};
```

Each socket contains two data queues, *so_rcv* and *so_snd*, and a pointer to routines which provide supporting services. The type of the socket, *so_type* is defined at socket creation time and used in selecting those services which are appropriate to support it. The supporting protocol is selected at socket creation time and recorded in the socket data structure for later use. Protocols are defined by a table of procedures, the *protosw* structure, which will be described in detail later. A pointer to a protocol specific data structure, the "protocol control block" is also present in the socket structure. Protocols control this data structure and it normally includes a back pointer to the parent socket structure(s) to allow easy lookup when returning information to a user (for example, placing an error number in the *so_error* field). The other entries in the socket structure are used in queueing connection requests, validating user requests, storing socket characteristics (e.g. options supplied at the time a socket is created), and maintaining a socket's state.

Processes "rendezvous at a socket" in many instances. For instance, when a process wishes to extract data from a socket's receive queue and it is empty, or lacks sufficient data to satisfy the request, the process blocks, supplying the address of the receive queue as an "wait channel' to be used in notification. When data arrives for the process and is placed in the socket's queue, the blocked process is identified by the fact it is waiting "on the queue".

6.1.1. Socket state

A socket's state is defined from the following:

```
#define  SS_NOFDREF          0x001   /* no file table ref any more */
#define  SS_ISCONNECTED      0x002   /* socket connected to a peer */
#define  SS_ISCONNECTING     0x004   /* in process of connecting to peer */
#define  SS_ISDISCONNECTING  0x008   /* in process of disconnecting */
#define  SS_CANTSENDMORE     0x010   /* can't send more data to peer */
#define  SS_CANTRCVMORE      0x020   /* can't receive more data from peer */
#define  SS_CONNAWAITING     0x040   /* connections awaiting acceptance */
#define  SS_RCVATMARK        0x080   /* at mark on input */

#define  SS_PRIV             0x100   /* privileged */
#define  SS_NBIO             0x200   /* non-blocking ops */
#define  SS_ASYNC            0x400   /* async i/o notify */
```

The state of a socket is manipulated both by the protocols and the user (through system calls). When a socket is created the state is defined based on the type of input/output the user wishes to perform. "Non-blocking" I/O implies a process should never be blocked to await resources. Instead, any call which would block returns prematurely with the error EWOULDBLOCK (the service request may be partially fulfilled, e.g. a request for more data than is present).

If a process requested "asynchronous" notification of events related to the socket the SIGIO signal is posted to the process. An event is a change in the socket's state, examples of such occurances are: space becoming available in the send queue, new data available in the receive queue, connection establishment or disestablishment, etc.

A socket may be marked "priviledged" if it was created by the super-user. Only priviledged sockets may send broadcast packets, or bind addresses in priviledged portions of an address space.

6.1.2. Socket data queues

A socket's data queue contains a pointer to the data stored in the queue and other entries related to the management of the data. The following structure defines a data queue:

```
struct sockbuf {
        short       sb_cc;          /* actual chars in buffer */
        short       sb_hiwat;       /* max actual char count */
        short       sb_mbcnt;       /* chars of mbufs used */
        short       sb_mbmax;       /* max chars of mbufs to use */
        short       sb_lowat;       /* low water mark */
        short       sb_timeo;       /* timeout */
        struct      mbuf *sb_mb;    /* the mbuf chain */
        struct      proc *sb_sel;   /* process selecting read/write */
        short       sb_flags;       /* flags, see below */
};
```

Data is stored in a queue as a chain of mbufs. The actual count of characters as well as high and low water marks are used by the protocols in controlling the flow of data. The socket routines cooperate in implementing the flow control policy by blocking a process when it requests to send data and the high water mark has been reached, or when it requests to receive data and less than the low water mark is present (assuming non-blocking I/O has not been specified).

When a socket is created, the supporting protocol "reserves" space for the send and receive queues of the socket. The actual storage associated with a socket queue may fluctuate during a socket's lifetime, but is assumed this reservation will always allow a protocol to acquire enough memory to satisfy the high water marks.

The timeout and select values are manipulated by the socket routines in implementing various portions of the interprocess communications facilities and will not be described here.

A socket queue has a number of flags used in synchronizing access to the data and in acquiring resources;

```
#define  SB_LOCK     0x01   /* lock on data queue (so_rcv only) */
#define  SB_WANT     0x02   /* someone is waiting to lock */
#define  SB_WAIT     0x04   /* someone is waiting for data/space */
#define  SB_SEL      0x08   /* buffer is selected */
#define  SB_COLL     0x10   /* collision selecting */
```

The last two flags are manipulated by the system in implementing the select mechanism.

6.1.3. Socket connection queueing

In dealing with connection oriented sockets (e.g. SOCK_STREAM) the two sides are considered distinct. One side is termed *active*, and generates connection requests. The other side is called *passive* and accepts connection requests.

From the passive side, a socket is created with the option SO_ACCEPTCONN specified, creating two queues of sockets: *so_q0* for connections in progress and *so_q* for connections already made and awaiting user acceptance. As a protocol is preparing incoming connections, it creates a socket structure queued on *so_q0* by calling the routine *sonewconn*(). When the connection is established, the socket structure is then transfered to *so_q*, making it available for an accept.

If an SO_ACCEPTCONN socket is closed with sockets on either *so_q0* or *so_q*, these sockets are dropped.

6.2. Protocol layer(s)

Protocols are described by a set of entry points and certain socket visible characteristics, some of which are used in deciding which socket type(s) they may support.

An entry in the "protocol switch" table exists for each protocol module configured into the system. It has the following form:

```
struct protosw {
        short       pr_type;            /* socket type used for */
        short       pr_family;          /* protocol family */
        short       pr_protocol;        /* protocol number */
        short       pr_flags;           /* socket visible attributes */
/* protocol-protocol hooks */
        int         (*pr_input)();      /* input to protocol (from below) */
        int         (*pr_output)();     /* output to protocol (from above) */
        int         (*pr_ctlinput)();   /* control input (from below) */
        int         (*pr_ctloutput)();  /* control output (from above) */
/* user-protocol hook */
        int         (*pr_usrreq)();     /* user request */
/* utility hooks */
        int         (*pr_init)();       /* initialization routine */
        int         (*pr_fasttimo)();   /* fast timeout (200ms) */
        int         (*pr_slowtimo)();   /* slow timeout (500ms) */
        int         (*pr_drain)();      /* flush any excess space possible */
};
```

A protocol is called through the *pr_init* entry before any other. Thereafter it is called every 200 milliseconds through the *pr_fasttimo* entry and every 500 milliseconds through the *pr_slowtimo* for timer based actions. The system will call the *pr_drain* entry if it is low on space and this should throw away any non-critical data.

Protocols pass data between themselves as chains of mbufs using the *pr_input* and *pr_output* routines. *Pr_input* passes data up (towards the user) and *pr_output* passes it down (towards the network); control information passes up and down on *pr_ctlinput* and *pr_ctloutput*. The protocol is responsible for the space occupied by any the arguments to these entries and must dispose of it.

The *pr_userreq* routine interfaces protocols to the socket code and is described below.

The *pr_flags* field is constructed from the following values:

```
#define PR_ATOMIC          0x01   /* exchange atomic messages only */
#define PR_ADDR            0x02   /* addresses given with messages */
#define PR_CONNREQUIRED    0x04   /* connection required by protocol */
#define PR_WANTRCVD        0x08   /* want PRU_RCVD calls */
#define PR_RIGHTS          0x10   /* passes capabilities */
```

Protocols which are connection-based specify the PR_CONNREQUIRED flag so that the socket routines will never attempt to send data before a connection has been established. If the PR_WANTRCVD flag is set, the socket routines will notfiy the protocol when the user has removed data from the socket's receive queue. This allows the protocol to implement acknowledgement on user receipt, and also update windowing information based on the amount of space available in the receive queue. The PR_ADDR field indicates any data placed in the socket's receive queue will be preceded by the address of the sender. The PR_ATOMIC flag specifies each *user* request to send data must be performed in a single *protocol* send request; it is the protocol's responsibility to maintain record boundaries on data to be sent. The PR_RIGHTS flag indicates the protocol supports the passing of capabilities; this is currently used only the protocols in the UNIX protocol family.

When a socket is created, the socket routines scan the protocol table looking for an appropriate protocol to support the type of socket being created. The *pr_type* field contains one of the possible socket types (e.g. SOCK_STREAM), while the *pr_family* field indicates which protocol family the protocol belongs to. The *pr_protocol* field contains the protocol number of the protocol, normally a well known value.

6.3. Network-interface layer

Each network-interface configured into a system defines a path through which packets may be sent and received. Normally a hardware device is associated with this interface, though there is no requirement for this (for example, all systems have a software "loopback" interface used for debugging and performance analysis). In addition to manipulating the hardware device, an interface module is responsible for encapsulation and deencapsulation of any low level header information required to deliver a message to it's destination. The selection of which interface to use in delivering packets is a routing decision carried out at a higher level than the network-interface layer. Each interface normally identifies itself at boot time to the routing module so that it may be selected for packet delivery.

An interface is defined by the following structure,

```
struct ifnet {
        char        *if_name;           /* name, e.g. "en" or "lo" */
        short       if_unit;            /* sub-unit for lower level driver */
        short       if_mtu;             /* maximum transmission unit */
        int         if_net;             /* network number of interface */
        short       if_flags;           /* up/down, broadcast, etc. */
        short       if_timer;           /* time 'til if_watchdog called */
        int         if_host[2];         /* local net host number */
        struct      sockaddr if_addr;   /* address of interface */
        union {
                    struct              sockaddr ifu_broadaddr;
                    struct              sockaddr ifu_dstaddr;
        } if_ifu;
        struct      ifqueue if_snd;     /* output queue */
        int         (*if_init)();       /* init routine */
        int         (*if_output)();     /* output routine */
        int         (*if_ioctl)();      /* ioctl routine */
        int         (*if_reset)();      /* bus reset routine */
        int         (*if_watchdog)();   /* timer routine */
        int         if_ipackets;        /* packets received on interface */
        int         if_ierrors;         /* input errors on interface */
        int         if_opackets;        /* packets sent on interface */
        int         if_oerrors;         /* output errors on interface */
        int         if_collisions;      /* collisions on csma interfaces */
        struct      ifnet *if_next;
};
```

Each interface has a send queue and routines used for initialization, *if_init*, and output, *if_output*. If the interface resides on a system bus, the routine *if_reset* will be called after a bus reset has been performed. An interface may also specify a timer routine, *if_watchdog*, which should be called every *if_timer* seconds (if non-zero).

The state of an interface and certain characteristics are stored in the *if_flags* field. The following values are possible:

```
#define  IFF_UP             0x1    /* interface is up */
#define  IFF_BROADCAST      0x2    /* broadcast address valid */
#define  IFF_DEBUG          0x4    /* turn on debugging */
#define  IFF_ROUTE          0x8    /* routing entry installed */
#define  IFF_POINTOPOINT    0x10   /* interface is point-to-point link */
#define  IFF_NOTRAILERS     0x20   /* avoid use of trailers */
#define  IFF_RUNNING        0x40   /* resources allocated */
#define  IFF_NOARP          0x80   /* no address resolution protocol */
```

If the interface is connected to a network which supports transmission of *broadcast* packets, the IFF_BROADCAST flag will be set and the *if_broadaddr* field will contain the address to be used in sending or accepting a broadcast packet. If the interface is associated with a point to point hardware link (for example, a DEC DMR-11), the IFF_POINTOPOINT flag will be set and *if_dstaddr* will contain the address of the host on the other side of the connection. These addresses and the local address of the interface, *if_addr*, are used in filtering incoming packets. The interface sets IFF_RUNNING after it has allocated system resources and posted an initial read on the device it manages. This state bit is used to avoid multiple allocation requests when an interface's address is changed. The IFF_NOTRAILERS flag indicates the interface should refrain from using a *trailer* encapsulation on outgoing packets; *trailer* protocols are described in section 14. The IFF_NOARP flag indicates the interface should not use an "address resolution protocol" in mapping internetwork addresses to local network addresses.

The information stored in an *ifnet* structure for point to point communication devices is not currently used by the system internally. Rather, it is used by the user level routing process in determining host network connections and in initially devising routes (refer to chapter 10 for more information).

Various statistics are also stored in the interface structure. These may be viewed by users using the *netstat*(1) program.

The interface address and flags may be set with the SIOCSIFADDR and SIOCSIFFLAGS ioctls. SIOCSIFADDR is used to initially define each interface's address; SIOGSIFFLAGS can be used to mark an interface down and perform site-specific configuration.

6.3.1. UNIBUS interfaces

All hardware related interfaces currently reside on the UNIBUS. Consequently a common set of utility routines for dealing with the UNIBUS has been developed. Each UNIBUS interface utilizes a structure of the following form:

```
struct  ifuba {
        short       ifu_uban;                   /* uba number */
        short       ifu_hlen;                   /* local net header length */
        struct      uba_regs *ifu_uba;          /* uba regs, in vm */
        struct ifrw {
                    caddr_t     ifrw_addr;      /* virt addr of header */
                    int         ifrw_bdp;       /* unibus bdp */
                    int         ifrw_info;      /* value from ubaalloc */
                    int         ifrw_proto;     /* map register prototype */
                    struct      pte *ifrw_mr;/* base of map registers */
        } ifu_r, ifu_w;
        struct      pte ifu_wmap[IF_MAXNUBAMR];/* base pages for output */
        short       ifu_xswapd;                 /* mask of clusters swapped */
        short       ifu_flags;                  /* used during uballoc's */
        struct      mbuf *ifu_xtofree;          /* pages being dma'd out */
};
```

The *if_uba* structure describes UNIBUS resources held by an interface. IF_NUBAMR map registers are held for datagram data, starting at *ifr_mr*. UNIBUS map register *ifr_mr*[-1] maps the local network header ending on a page boundary. UNIBUS data paths are reserved for read and for write, given by *ifr_bdp*. The prototype of the map registers for read and for write is saved in *ifr_proto*.

When write transfers are not full pages on page boundaries the data is just copied into the pages mapped on the UNIBUS and the transfer is started. If a write transfer is of a (1024 byte) page size and on a page boundary, UNIBUS page table entries are swapped to reference the pages, and then the initial pages are remapped from *ifu_wmap* when the transfer completes.

When read transfers give whole pages of data to be input, page frames are allocated from a network page list and traded with the pages already containing the data, mapping the allocated pages to replace the input pages for the next UNIBUS data input.

The following utility routines are available for use in writing network interface drivers, all use the *ifuba* structure described above.

if_ubainit(ifu, uban, hlen, nmr);

if_ubainit allocates resources on UNIBUS adaptor *uban* and stores the resultant information in the *ifuba* structure pointed to by *ifu*. It is called only at boot time or after a UNIBUS reset. Two data paths (buffered or unbuffered, depending on the *ifu_flags* field) are allocated, one for reading and one for writing. The *nmr* parameter indicates the number of UNIBUS mapping registers required to map a maximal sized packet onto the UNIBUS, while *hlen* specifies the size of a local network header, if any, which should be mapped separately from

the data (see the description of trailer protocols in chapter 14). Sufficient UNIBUS mapping registers and pages of memory are allocated to initialize the input data path for an initial read. For the output data path, mapping registers and pages of memory are also allocated and mapped onto the UNIBUS. The pages associated with the output data path are held in reserve in the event a write requires copying non-page-aligned data (see *if_wubaput* below). If *if_ubainit* is called with resources already allocated, they will be used instead of allocating new ones (this normally occurs after a UNIBUS reset). A 1 is returned when allocation and initialization is successful, 0 otherwise.

m = if_rubaget(ifu, totlen, off0);

if_rubaget pulls read data off an interface. *totlen* specifies the length of data to be obtained, not counting the local network header. If *off0* is non-zero, it indicates a byte offset to a trailing local network header which should be copied into a separate mbuf and prepended to the front of the resultant mbuf chain. When page sized units of data are present and are page-aligned, the previously mapped data pages are remapped into the mbufs and swapped with fresh pages; thus avoiding any copying. A 0 return value indicates a failure to allocate resources.

if_wubaput(ifu, m);

if_wubaput maps a chain of mbufs onto a network interface in preparation for output. The chain includes any local network header, which is copied so that it resides in the mapped and aligned I/O space. Any other mbufs which contained non page sized data portions are also copied to the I/O space. Pages mapped from a previous output operation (no longer needed) are unmapped and returned to the network page pool.

7. Socket/protocol interface

The interface between the socket routines and the communication protocols is through the *pr_usrreq* routine defined in the protocol switch table. The following requests to a protocol module are possible:

```
#define  PRU_ATTACH          0     /* attach protocol */
#define  PRU_DETACH          1     /* detach protocol */
#define  PRU_BIND            2     /* bind socket to address */
#define  PRU_LISTEN          3     /* listen for connection */
#define  PRU_CONNECT         4     /* establish connection to peer */
#define  PRU_ACCEPT          5     /* accept connection from peer */
#define  PRU_DISCONNECT      6     /* disconnect from peer */
#define  PRU_SHUTDOWN        7     /* won't send any more data */
#define  PRU_RCVD            8     /* have taken data; more room now */
#define  PRU_SEND            9     /* send this data */
#define  PRU_ABORT           10    /* abort (fast DISCONNECT, DETATCH) */
#define  PRU_CONTROL         11    /* control operations on protocol */
#define  PRU_SENSE           12    /* return status into m */
#define  PRU_RCVOOB          13    /* retrieve out of band data */
#define  PRU_SENDOOB         14    /* send out of band data */
#define  PRU_SOCKADDR        15    /* fetch socket's address */
#define  PRU_PEERADDR        16    /* fetch peer's address */
#define  PRU_CONNECT2        17    /* connect two sockets */
/* begin for protocols internal use */
#define  PRU_FASTTIMO        18    /* 200ms timeout */
#define  PRU_SLOWTIMO        19    /* 500ms timeout */
#define  PRU_PROTORCV        20    /* receive from below */
#define  PRU_PROTOSEND       21    /* send to below */
```

A call on the user request routine is of the form,

```
error = (*protosw[].pr_usrreq)(up, req, m, addr, rights);
int error; struct socket *up; int req; struct mbuf *m, *rights; caddr_t addr;
```

The mbuf chain, *m*, and the address are optional parameters. The *rights* parameter is an optional pointer to an mbuf chain containing user specified capabilities (see the *sendmsg* and *recvmsg* system calls). The protocol is responsible for disposal of both mbuf chains. A non-zero return value gives a UNIX error number which should be passed to higher level software. The following paragraphs describe each of the requests possible.

PRU_ATTACH

When a protocol is bound to a socket (with the *socket* system call) the protocol module is called with this request. It is the responsibility of the protocol module to allocate any resources necessary. The "attach" request will always precede any of the other requests, and should not occur more than once.

PRU_DETACH

This is the antithesis of the attach request, and is used at the time a socket is deleted. The protocol module may deallocate any resources assigned to the socket.

PRU_BIND

When a socket is initially created it has no address bound to it. This request indicates an address should be bound to an existing socket. The protocol module must verify the requested address is valid and available for use.

PRU_LISTEN

The "listen" request indicates the user wishes to listen for incoming connection requests on the associated socket. The protocol module should perform any state changes needed to carry out this request (if possible). A "listen" request always precedes a request to accept a

connection.

PRU_CONNECT

The "connect" request indicates the user wants to a establish an association. The *addr* parameter supplied describes the peer to be connected to. The effect of a connect request may vary depending on the protocol. Virtual circuit protocols, such as TCP [Postel80b], use this request to initiate establishment of a TCP connection. Datagram protocols, such as UDP [Postel79], simply record the peer's address in a private data structure and use it to tag all outgoing packets. There are no restrictions on how many times a connect request may be used after an attach. If a protocol supports the notion of *multi-casting*, it is possible to use multiple connects to establish a multi-cast group. Alternatively, an association may be broken by a PRU_DISCONNECT request, and a new association created with a subsequent connect request; all without destroying and creating a new socket.

PRU_ACCEPT

Following a successful PRU_LISTEN request and the arrival of one or more connections, this request is made to indicate the user has accepted the first connection on the queue of pending connections. The protocol module should fill in the supplied address buffer with the address of the connected party.

PRU_DISCONNECT

Eliminate an association created with a PRU_CONNECT request.

PRU_SHUTDOWN

This call is used to indicate no more data will be sent and/or received (the *addr* parameter indicates the direction of the shutdown, as encoded in the *soshutdown* system call). The protocol may, at its discretion, deallocate any data structures related to the shutdown.

PRU_RCVD

This request is made only if the protocol entry in the protocol switch table includes the PR_WANTRCVD flag. When a user removes data from the receive queue this request will be sent to the protocol module. It may be used to trigger acknowledgements, refresh windowing information, initiate data transfer, etc.

PRU_SEND

Each user request to send data is translated into one or more PRU_SEND requests (a protocol may indicate a single user send request must be translated into a single PRU_SEND request by specifying the PR_ATOMIC flag in its protocol description). The data to be sent is presented to the protocol as a list of mbufs and an address is, optionally, supplied in the *addr* parameter. The protocol is responsible for preserving the data in the socket's send queue if it is not able to send it immediately, or if it may need it at some later time (e.g. for retransmission).

PRU_ABORT

This request indicates an abnormal termination of service. The protocol should delete any existing association(s).

PRU_CONTROL

The "control" request is generated when a user performs a UNIX *ioctl* system call on a socket (and the ioctl is not intercepted by the socket routines). It allows protocol-specific operations to be provided outside the scope of the common socket interface. The *addr* parameter contains a pointer to a static kernel data area where relevant information may be obtained or returned. The *m* parameter contains the actual *ioctl* request code (note the non-standard calling convention).

PRU_SENSE

The "sense" request is generated when the user makes an *fstat* system call on a socket; it requests status of the associated socket. There currently is no common format for the status returned. Information which might be returned includes per-connection statistics, protocol state, resources currently in use by the connection, the optimal transfer size for the connection (based on windowing information and maximum packet size). The *addr* parameter

contains a pointer to a static kernel data area where the status buffer should be placed.

PRU_RCVOOB
: Any "out-of-band" data presently available is to be returned. An mbuf is passed in to the protocol module and the protocol should either place data in the mbuf or attach new mbufs to the one supplied if there is insufficient space in the single mbuf.

PRU_SENDOOB
: Like PRU_SEND, but for out-of-band data.

PRU_SOCKADDR
: The local address of the socket is returned, if any is currently bound to the it. The address format (protocol specific) is returned in the *addr* parameter.

PRU_PEERADDR
: The address of the peer to which the socket is connected is returned. The socket must be in a SS_ISCONNECTED state for this request to be made to the protocol. The address format (protocol specific) is returned in the *addr* parameter.

PRU_CONNECT2
: The protocol module is supplied two sockets and requested to establish a connection between the two without binding any addresses, if possible. This call is used in implementing the system call.

The following requests are used internally by the protocol modules and are never generated by the socket routines. In certain instances, they are handed to the *pr_usrreq* routine solely for convenience in tracing a protocol's operation (e.g. PRU_SLOWTIMO).

PRU_FASTTIMO
: A "fast timeout" has occured. This request is made when a timeout occurs in the protocol's *pr_fastimo* routine. The *addr* parameter indicates which timer expired.

PRU_SLOWTIMO
: A "slow timeout" has occured. This request is made when a timeout occurs in the protocol's *pr_slowtimo* routine. The *addr* parameter indicates which timer expired.

PRU_PROTORCV
: This request is used in the protocol-protocol interface, not by the routines. It requests reception of data destined for the protocol and not the user. No protocols currently use this facility.

PRU_PROTOSEND
: This request allows a protocol to send data destined for another protocol module, not a user. The details of how data is marked "addressed to protocol" instead of "addressed to user" are left to the protocol modules. No protocols currently use this facility.

8. Protocol/protocol interface

The interface between protocol modules is through the *pr_usrreq*, *pr_input*, *pr_output*, *pr_ctlinput*, and *pr_ctloutput* routines. The calling conventions for all but the *pr_usrreq* routine are expected to be specific to the protocol modules and are not guaranteed to be consistent across protocol families. We will examine the conventions used for some of the Internet protocols in this section as an example.

8.1. pr_output

The Internet protocol UDP uses the convention,

```
error = udp_output(inp, m);
int error; struct inpcb *inp; struct mbuf *m;
```

where the *inp*, "*in*ternet *p*rotocol *c*ontrol *b*lock", passed between modules conveys per connection state information, and the mbuf chain contains the data to be sent. UDP performs consistency checks, appends its header, calculates a checksum, etc. before passing the packet on to the IP module:

```
error = ip_output(m, opt, ro, allowbroadcast);
int error; struct mbuf *m, *opt; struct route *ro; int allowbroadcast;
```

The call to IP's output routine is more complicated than that for UDP, as befits the additional work the IP module must do. The *m* parameter is the data to be sent, and the *opt* parameter is an optional list of IP options which should be placed in the IP packet header. The *ro* parameter is is used in making routing decisions (and passing them back to the caller). The final parameter, *allowbroadcast* is a flag indicating if the user is allowed to transmit a broadcast packet. This may be inconsequential if the underlying hardware does not support the notion of broadcasting.

All output routines return 0 on success and a UNIX error number if a failure occured which could be immediately detected (no buffer space available, no route to destination, etc.).

8.2. pr_input

Both UDP and TCP use the following calling convention,

```
(void) (*protosw[].pr_input)(m);
struct mbuf *m;
```

Each mbuf list passed is a single packet to be processed by the protocol module.

The IP input routine is a VAX software interrupt level routine, and so is not called with any parameters. It instead communicates with network interfaces through a queue, *ipintrq*, which is identical in structure to the queues used by the network interfaces for storing packets awaiting transmission.

8.3. pr_ctlinput

This routine is used to convey "control" information to a protocol module (i.e. information which might be passed to the user, but is not data). This routine, and the *pr_ctloutput* routine, have not been extensively developed, and thus suffer from a "clumsiness" that can only be improved as more demands are placed on it.

The common calling convention for this routine is,

```
(void) (*protosw[].pr_ctlinput)(req, info);
int req; caddr_t info;
```

The *req* parameter is one of the following,

```
#define PRC_IFDOWN              0   /* interface transition */
#define PRC_ROUTEDEAD           1   /* select new route if possible */
#define PRC_QUENCH              4   /* some said to slow down */
#define PRC_HOSTDEAD            6   /* normally from IMP */
#define PRC_HOSTUNREACH         7   /* ditto */
#define PRC_UNREACH_NET         8   /* no route to network */
#define PRC_UNREACH_HOST        9   /* no route to host */
#define PRC_UNREACH_PROTOCOL    10  /* dst says bad protocol */
#define PRC_UNREACH_PORT        11  /* bad port # */
#define PRC_MSGSIZE             12  /* message size forced drop */
#define PRC_REDIRECT_NET        13  /* net routing redirect */
#define PRC_REDIRECT_HOST       14  /* host routing redirect */
#define PRC_TIMXCEED_INTRANS    17  /* packet lifetime expired in transit */
#define PRC_TIMXCEED_REASS      18  /* lifetime expired on reass q */
#define PRC_PARAMPROB           19  /* header incorrect */
```

while the *info* parameter is a "catchall" value which is request dependent. Many of the requests have obviously been derived from ICMP (the Internet Control Message Protocol), and from error messages defined in the 1822 host/IMP convention [BBN78]. Mapping tables exist to convert control requests to UNIX error codes which are delivered to a user.

8.4. pr_ctloutput

This routine is not currently used by any protocol modules.

9. Protocol/network-interface interface

The lowest layer in the set of protocols which comprise a protocol family must interface itself to one or more network interfaces in order to transmit and receive packets. It is assumed that any routing decisions have been made before handing a packet to a network interface, in fact this is absolutely necessary in order to locate any interface at all (unless, of course, one uses a single "hardwired" interface). There are two cases to be concerned with, transmission of a packet, and receipt of a packet; each will be considered separately.

9.1. Packet transmission

Assuming a protocol has a handle on an interface, *ifp*, a (struct ifnet *), it transmits a fully formatted packet with the following call,

```
error = (*ifp->if_output)(ifp, m, dst)
int error; struct ifnet *ifp; struct mbuf *m; struct sockaddr *dst;
```

The output routine for the network interface transmits the packet *m* to the *dst* address, or returns an error indication (a UNIX error number). In reality transmission may not be immediate, or successful; normally the output routine simply queues the packet on its send queue and primes an interrupt driven routine to actually transmit the packet. For unreliable mediums, such as the Ethernet, "successful" transmission simply means the packet has been placed on the cable without a collision. On the other hand, an 1822 interface guarantees proper delivery or an error indication for each message transmitted. The model employed in the networking system attaches no promises of delivery to the packets handed to a network interface, and thus corresponds more closely to the Ethernet. Errors returned by the output routine are normally trivial in nature (no buffer space, address format not handled, etc.).

9.2. Packet reception

Each protocol family must have one or more "lowest level" protocols. These protocols deal with internetwork addressing and are responsible for the delivery of incoming packets to the proper protocol processing modules. In the PUP model [Boggs78] these protocols are termed Level 1 protocols, in the ISO model, network layer protocols. In our system each such protocol module has an input packet queue assigned to it. Incoming packets received by a network interface are queued up for the protocol module and a VAX software interrupt is posted to initiate processing.

Three macros are available for queueing and dequeueing packets,

IF_ENQUEUE(ifq, m)
: This places the packet *m* at the tail of the queue *ifq*.

IF_DEQUEUE(ifq, m)
: This places a pointer to the packet at the head of queue *ifq* in *m*. A zero value will be returned in *m* if the queue is empty.

IF_PREPEND(ifq, m)
: This places the packet *m* at the head of the queue *ifq*.

Each queue has a maximum length associated with it as a simple form of congestion control. The macro IF_QFULL(ifq) returns 1 if the queue is filled, in which case the macro IF_DROP(ifq) should be used to bump a count of the number of packets dropped and the offending packet dropped. For example, the following code fragment is commonly found in a network interface's input routine,

```
if (IF_QFULL(inq)) {
        IF_DROP(inq);
        m_freem(m);
} else
        IF_ENQUEUE(inq, m);
```

10. Gateways and routing issues

The system has been designed with the expectation that it will be used in an internetwork environment. The "canonical" environment was envisioned to be a collection of local area networks connected at one or more points through hosts with multiple network interfaces (one on each local area network), and possibly a connection to a long haul network (for example, the ARPANET). In such an environment, issues of gatewaying and packet routing become very important. Certain of these issues, such as congestion control, have been handled in a simplistic manner or specifically not addressed. Instead, where possible, the network system attempts to provide simple mechanisms upon which more involved policies may be implemented. As some of these problems become better understood, the solutions developed will be incorporated into the system.

This section will describe the facilities provided for packet routing. The simplistic mechanisms provided for congestion control are described in chapter 12.

10.1. Routing tables

The network system maintains a set of routing tables for selecting a network interface to use in delivering a packet to its destination. These tables are of the form:

```
struct rtentry {
        u_long      rt_hash;        /* hash key for lookups */
        struct      sockaddr rt_dst;    /* destination net or host */
        struct      sockaddr rt_gateway;/* forwarding agent */
        short       rt_flags;       /* see below */
        short       rt_refcnt;      /* no. of references to structure */
        u_long      rt_use;         /* packets sent using route */
        struct      ifnet *rt_ifp;      /* interface to give packet to */
};
```

The routing information is organized in two separate tables, one for routes to a host and one for routes to a network. The distinction between hosts and networks is necessary so that a single mechanism may be used for both broadcast and multi-drop type networks, and also for networks built from point-to-point links (e.g DECnet [DEC80]).

Each table is organized as a hashed set of linked lists. Two 32-bit hash values are calculated by routines defined for each address family; one based on the destination being a host, and one assuming the target is the network portion of the address. Each hash value is used to locate a hash chain to search (by taking the value modulo the hash table size) and the entire 32-bit value is then used as a key in scanning the list of routes. Lookups are applied first to the routing table for hosts, then to the routing table for networks. If both lookups fail, a final lookup is made for a "wildcard" route (by convention, network 0). By doing this, routes to a specific host on a network may be present as well as routes to the network. This also allows a "fall back" network route to be defined to an "smart" gateway which may then perform more intelligent routing.

Each routing table entry contains a destination (who's at the other end of the route), a gateway to send the packet to, and various flags which indicate the route's status and type (host or network). A count of the number of packets sent using the route is kept for use in deciding between multiple routes to the same destination (see below), and a count of "held references" to the dynamically allocated structure is maintained to insure memory reclamation occurs only when the route is not in use. Finally a pointer to the a network interface is kept; packets sent using the route should be handed to this interface.

Routes are typed in two ways: either as host or network, and as "direct" or "indirect". The host/network distinction determines how to compare the *rt_dst* field during lookup. If the route is to a network, only a packet's destination network is compared to the *rt_dst* entry stored in the table. If the route is to a host, the addresses must match bit for bit.

The distinction between "direct" and "indirect" routes indicates whether the destination is directly connected to the source. This is needed when performing local network encapsulation. If

a packet is destined for a peer at a host or network which is not directly connected to the source, the internetwork packet header will indicate the address of the eventual destination, while the local network header will indicate the address of the intervening gateway. Should the destination be directly connected, these addresses are likely to be identical, or a mapping between the two exists. The RTF_GATEWAY flag indicates the route is to an "indirect" gateway agent and the local network header should be filled in from the *rt_gateway* field instead of *rt_dst*, or from the internetwork destination address.

It is assumed multiple routes to the same destination will not be present unless they are deemed *equal* in cost (the current routing policy process never installs multiple routes to the same destination). However, should multiple routes to the same destination exist, a request for a route will return the "least used" route based on the total number of packets sent along this route. This can result in a "ping-pong" effect (alternate packets taking alternate routes), unless protocols "hold onto" routes until they no longer find them useful; either because the destination has changed, or because the route is lossy.

Routing redirect control messages are used to dynamically modify existing routing table entries as well as dynamically create new routing table entries. On hosts where exhaustive routing information is too expensive to maintain (e.g. work stations), the combination of wildcard routing entries and routing redirect messages can be used to provide a simple routing management scheme without the use of a higher level policy process. Statistics are kept by the routing table routines on the use of routing redirect messages and their affect on the routing tables. These statistics may be viewed using

Status information other than routing redirect control messages may be used in the future, but at present they are ignored. Likewise, more intelligent "metrics" may be used to describe routes in the future, possibly based on bandwidth and monetary costs.

10.2. Routing table interface

A protocol accesses the routing tables through three routines, one to allocate a route, one to free a route, and one to process a routing redirect control message. The routine *rtalloc* performs route allocation; it is called with a pointer to the following structure,

```
struct route {
        struct      rtentry *ro_rt;
        struct      sockaddr ro_dst;
};
```

The route returned is assumed "held" by the caller until disposed of with an *rtfree* call. Protocols which implement virtual circuits, such as TCP, hold onto routes for the duration of the circuit's lifetime, while connection-less protocols, such as UDP, currently allocate and free routes on each transmission.

The routine *rtredirect* is called to process a routing redirect control message. It is called with a destination address and the new gateway to that destination. If a non-wildcard route exists to the destination, the gateway entry in the route is modified to point at the new gateway supplied. Otherwise, a new routing table entry is inserted reflecting the information supplied. Routes to interfaces and routes to gateways which are not directly accesible from the host are ignored.

10.3. User level routing policies

Routing policies implemented in user processes manipulate the kernel routing tables through two *ioctl* calls. The commands SIOCADDRT and SIOCDELRT add and delete routing entries, respectively; the tables are read through the /dev/kmem device. The decision to place policy decisions in a user process implies routing table updates may lag a bit behind the identification of new routes, or the failure of existing routes, but this period of instability is normally very small with proper implementation of the routing process. Advisory information, such as ICMP error messages and IMP diagnostic messages, may be read from raw sockets (described in the next section).

One routing policy process has already been implemented. The system standard "routing daemon" uses a variant of the Xerox NS Routing Information Protocol [Xerox82] to maintain up to date routing tables in our local environment. Interaction with other existing routing protocols, such as the Internet GGP (Gateway-Gateway Protocol), may be accomplished using a similar process.

11. Raw sockets

A raw socket is a mechanism which allows users direct access to a lower level protocol. Raw sockets are intended for knowledgeable processes which wish to take advantage of some protocol feature not directly accessible through the normal interface, or for the development of new protocols built atop existing lower level protocols. For example, a new version of TCP might be developed at the user level by utilizing a raw IP socket for delivery of packets. The raw IP socket interface attempts to provide an identical interface to the one a protocol would have if it were resident in the kernel.

The raw socket support is built around a generic raw socket interface, and (possibly) augmented by protocol-specific processing routines. This section will describe the core of the raw socket interface.

11.1. Control blocks

Every raw socket has a protocol control block of the following form,

```
struct rawcb {
        struct      rawcb *rcb_next;        /* doubly linked list */
        struct      rawcb *rcb_prev;
        struct      socket *rcb_socket;     /* back pointer to socket */
        struct      sockaddr rcb_faddr;     /* destination address */
        struct      sockaddr rcb_laddr;     /* socket's address */
        caddr_t     rcb_pcb;                /* protocol specific stuff */
        short       rcb_flags;
};
```

All the control blocks are kept on a doubly linked list for performing lookups during packet dispatch. Associations may be recorded in the control block and used by the output routine in preparing packets for transmission. The addresses are also used to filter packets on input; this will be described in more detail shortly. If any protocol specific information is required, it may be attached to the control block using the *rcb_pcb* field.

A raw socket interface is datagram oriented. That is, each send or receive on the socket requires a destination address. This address may be supplied by the user or stored in the control block and automatically installed in the outgoing packet by the output routine. Since it is not possible to determine whether an address is present or not in the control block, two flags, RAW_LADDR and RAW_FADDR, indicate if a local and foreign address are present. Another flag, RAW_DONTROUTE, indicates if routing should be performed on outgoing packets. If it is, a route is expected to be allocated for each "new" destination address. That is, the first time a packet is transmitted a route is determined, and thereafter each time the destination address stored in *rcb_route* differs from *rcb_faddr*, or *rcb_route.ro_rt* is zero, the old route is discarded and a new one allocated.

11.2. Input processing

Input packets are "assigned" to raw sockets based on a simple pattern matching scheme. Each network interface or protocol gives packets to the raw input routine with the call:

```
raw_input(m, proto, src, dst)
struct mbuf *m; struct sockproto *proto, struct sockaddr *src, *dst;
```

The data packet then has a generic header prepended to it of the form

```
struct raw_header {
        struct      sockproto raw_proto;
        struct      sockaddr raw_dst;
        struct      sockaddr raw_src;
};
```

and it is placed in a packet queue for the "raw input protocol" module. Packets taken from this queue are copied into any raw sockets that match the header according to the following rules,

1) The protocol family of the socket and header agree.
2) If the protocol number in the socket is non-zero, then it agrees with that found in the packet header.
3) If a local address is defined for the socket, the address format of the local address is the same as the destination address's and the two addresses agree bit for bit.
4) The rules of 3) are applied to the socket's foreign address and the packet's source address.

A basic assumption is that addresses present in the control block and packet header (as constructed by the network interface and any raw input protocol module) are in a canonical form which may be "block compared".

11.3. Output processing

On output the raw *pr_usrreq* routine passes the packet and raw control block to the raw protocol output routine for any processing required before it is delivered to the appropriate network interface. The output routine is normally the only code required to implement a raw socket interface.

12. Buffering and congestion control

One of the major factors in the performance of a protocol is the buffering policy used. Lack of a proper buffering policy can force packets to be dropped, cause falsified windowing information to be emitted by protocols, fragment host memory, degrade the overall host performance, etc. Due to problems such as these, most systems allocate a fixed pool of memory to the networking system and impose a policy optimized for "normal" network operation.

The networking system developed for UNIX is little different in this respect. At boot time a fixed amount of memory is allocated by the networking system. At later times more system memory may be requested as the need arises, but at no time is memory ever returned to the system. It is possible to garbage collect memory from the network, but difficult. In order to perform this garbage collection properly, some portion of the network will have to be "turned off" as data structures are updated. The interval over which this occurs must kept small compared to the average inter-packet arrival time, or too much traffic may be lost, impacting other hosts on the network, as well as increasing load on the interconnecting mediums. In our environment we have not experienced a need for such compaction, and thus have left the problem unresolved.

The mbuf structure was introduced in chapter 5. In this section a brief description will be given of the allocation mechanisms, and policies used by the protocols in performing connection level buffering.

12.1. Memory management

The basic memory allocation routines place no restrictions on the amount of space which may be allocated. Any request made is filled until the system memory allocator starts refusing to allocate additional memory. When the current quota of memory is insufficient to satisfy an mbuf allocation request, the allocator requests enough new pages from the system to satisfy the current request only. All memory owned by the network is described by a private page table used in remapping pages to be logically contiguous as the need arises. In addition, an array of reference counts parallels the page table and is used when multiple copies of a page are present.

Mbufs are 128 byte structures, 8 fitting in a 1Kbyte page of memory. When data is placed in mbufs, if possible, it is copied or remapped into logically contiguous pages of memory from the network page pool. Data smaller than the size of a page is copied into one or more 112 byte mbuf data areas.

12.2. Protocol buffering policies

Protocols reserve fixed amounts of buffering for send and receive queues at socket creation time. These amounts define the high and low water marks used by the socket routines in deciding when to block and unblock a process. The reservation of space does not currently result in any action by the memory management routines, though it is clear if one imposed an upper bound on the total amount of physical memory allocated to the network, reserving memory would become important.

Protocols which provide connection level flow control do this based on the amount of space in the associated socket queues. That is, send windows are calculated based on the amount of free space in the socket's receive queue, while receive windows are adjusted based on the amount of data awaiting transmission in the send queue. Care has been taken to avoid the "silly window syndrome" described in [Clark82] at both the sending and receiving ends.

12.3. Queue limiting

Incoming packets from the network are always received unless memory allocation fails. However, each Level 1 protocol input queue has an upper bound on the queue's length, and any packets exceeding that bound are discarded. It is possible for a host to be overwhelmed by excessive network traffic (for instance a host acting as a gateway from a high bandwidth network to a low bandwidth network). As a "defensive" mechanism the queue limits may be adjusted to throttle network traffic load on a host. Consider a host willing to devote some percentage of its machine to

handling network traffic. If the cost of handling an incoming packet can be calculated so that an acceptable "packet handling rate" can be determined, then input queue lengths may be dynamically adjusted based on a host's network load and the number of packets awaiting processing. Obviously, discarding packets is not a satisfactory solution to a problem such as this (simply dropping packets is likely to increase the load on a network); the queue lengths were incorporated mainly as a safeguard mechanism.

12.4. Packet forwarding

When packets can not be forwarded because of memory limitations, the system generates a "source quench" message. In addition, any other problems encountered during packet forwarding are also reflected back to the sender in the form of ICMP packets. This helps hosts avoid unneeded retransmissions.

Broadcast packets are never forwarded due to possible dire consequences. In an early stage of network development, broadcast packets were forwarded and a "routing loop" resulted in network saturation and every host on the network crashing.

13. Out of band data

Out of band data is a facility peculiar to the stream socket abstraction defined. Little agreement appears to exist as to what its semantics should be. TCP defines the notion of "urgent data" as in-line, while the NBS protocols [Burruss81] and numerous others provide a fully independent logical transmission channel along which out of band data is to be sent. In addition, the amount of the data which may be sent as an out of band message varies from protocol to protocol; everything from 1 bit to 16 bytes or more.

A stream socket's notion of out of band data has been defined as the lowest reasonable common denominator (at least reasonable in our minds); clearly this is subject to debate. Out of band data is expected to be transmitted out of the normal sequencing and flow control constraints of the data stream. A minimum of 1 byte of out of band data and one outstanding out of band message are expected to be supported by the protocol supporting a stream socket. It is a protocols perogative to support larger sized messages, or more than one outstanding out of band message at a time.

Out of band data is maintained by the protocol and usually not stored in the socket's send queue. The PRU_SENDOOB and PRU_RCVOOB requests to the *pr_usrreq* routine are used in sending and receiving data.

14. Trailer protocols

Core to core copies can be expensive. Consequently, a great deal of effort was spent in minimizing such operations. The VAX architecture provides virtual memory hardware organized in page units. To cut down on copy operations, data is kept in page sized units on page-aligned boundaries whenever possible. This allows data to be moved in memory simply by remapping the page instead of copying. The mbuf and network interface routines perform page table manipulations where needed, hiding the complexities of the VAX virtual memory hardware from higher level code.

Data enters the system in two ways: from the user, or from the network (hardware interface). When data is copied from the user's address space into the system it is deposited in pages (if sufficient data is present to fill an entire page). This encourages the user to transmit information in messages which are a multiple of the system page size.

Unfortunately, performing a similar operation when taking data from the network is very difficult. Consider the format of an incoming packet. A packet usually contains a local network header followed by one or more headers used by the high level protocols. Finally, the data, if any, follows these headers. Since the header information may be variable length, DMA'ing the eventual data for the user into a page aligned area of memory is impossible without a priori knowledge of the format (e.g. supporting only a single protocol header format).

To allow variable length header information to be present and still ensure page alignment of data, a special local network encapsulation may be used. This encapsulation, termed a *trailer protocol*, places the variable length header information after the data. A fixed size local network header is then prepended to the resultant packet. The local network header contains the size of the data portion, and a new *trailer protocol header*, inserted before the variable length information, contains the size of the variable length header information. The following trailer protocol header is used to store information regarding the variable length protocol header:

```
struct {
        short       protocol;       /* original protocol no. */
        short       length;         /* length of trailer */
};
```

The processing of the trailer protocol is very simple. On output, the local network header indicates a trailer encapsulation is being used. The protocol identifier also includes an indication of the number of data pages present (before the trailer protocol header). The trailer protocol header is initialized to contain the actual protocol and variable length header size, and appended to the data along with the variable length header information.

On input, the interface routines identify the trailer encapsulation by the protocol type stored in the local network header, then calculate the number of pages of data to find the beginning of the trailer. The trailing information is copied into a separate mbuf and linked to the front of the resultant packet.

Clearly, trailer protocols require cooperation between source and destination. In addition, they are normally cost effective only when sizable packets are used. The current scheme works because the local network encapsulation header is a fixed size, allowing DMA operations to be performed at a known offset from the first data page being received. Should the local network header be variable length this scheme fails.

Statistics collected indicate as much as 200Kb/s can be gained by using a trailer protocol with 1Kbyte packets. The average size of the variable length header was 40 bytes (the size of a minimal TCP/IP packet header). If hardware supports larger sized packets, even greater gains may be realized.

Acknowledgements

The internal structure of the system is patterned after the Xerox PUP architecture [Boggs79], while in certain places the Internet protocol family has had a great deal of influence in the design. The use of software interrupts for process invocation is based on similar facilities found in the VMS operating system. Many of the ideas related to protocol modularity, memory management, and network interfaces are based on Rob Gurwitz's TCP/IP implementation for the 4.1BSD version of UNIX on the VAX [Gurwitz81]. Greg Chesson explained his use of trailer encapsulations in Datakit, instigating their use in our system.

References

[Boggs79] Boggs, D. R., J. F. Shoch, E. A. Taft, and R. M. Metcalfe; *PUP: An Internetwork Architecture.* Report CSL-79-10. XEROX Palo Alto Research Center, July 1979.

[BBN78] Bolt Beranek and Newman; *Specification for the Interconnection of Host and IMP.* BBN Technical Report 1822. May 1978.

[Cerf78] Cerf, V. G.; The Catenet Model for Internetworking. Internet Working Group, IEN 48. July 1978.

[Clark82] Clark, D. D.; Window and Acknowledgement Strategy in TCP. Internet Working Group, IEN Draft Clark-2. March 1982.

[DEC80] Digital Equipment Corporation; *DECnet DIGITAL Network Architecture - General Description.* Order No. AA-K179A-TK. October 1980.

[Gurwitz81] Gurwitz, R. F.; VAX-UNIX Networking Support Project – Implementation Description. Internetwork Working Group, IEN 168. January 1981.

[ISO81] International Organization for Standardization. *ISO Open Systems Interconnection - Basic Reference Model.* ISO/TC 97/SC 16 N 719. August 1981.

[Joy82a] Joy, W.; Cooper, E.; Fabry, R.; Leffler, S.; and McKusick, M.; *4.2BSD System Manual.* Computer Systems Research Group, Technical Report 5. University of California, Berkeley. Draft of September 1, 1982.

[Postel79] Postel, J., ed. *DOD Standard User Datagram Protocol.* Internet Working Group, IEN 88. May 1979.

[Postel80a] Postel, J., ed. *DOD Standard Internet Protocol.* Internet Working Group, IEN 128. January 1980.

[Postel80b] Postel, J., ed. *DOD Standard Transmission Control Protocol.* Internet Working Group, IEN 129. January 1980.

[Xerox81] Xerox Corporation. *Internet Transport Protocols.* Xerox System Integration Standard 028112. December 1981.

[Zimmermann80] Zimmermann, H. OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection. IEEE Transactions on Communications. Com-28(4); 425-432. April 1980.

Disc Quotas in a UNIX* Environment.

Robert Elz

Department of Computer Science
University of Melbourne,
Parkville,
Victoria,
Australia.

ABSTRACT

In most computing environments, disc space is not infinite. The disc quota system provides a mechanism to control usage of disc space, on an individual basis.

Quotas may be set for each individual user, on any, or all filesystems.

The quota system will warn users when they exceed their allotted limit, but allow some extra space for current work. Repeatedly remaining over quota at logout, will cause a fatal over quota condition eventually.

The quota system is an optional part of VMUNIX that may be included when the system is configured.

5th July, 1983

* UNIX is a trademark of Bell Laboratories.

Disc Quotas in a UNIX* Environment.

Robert Elz

Department of Computer Science
University of Melbourne,
Parkville,
Victoria,
Australia.

1. Users' view of disc quotas

To most users, disc quotas will either be of no concern, or a fact of life that cannot be avoided. The *quota*(1) command will provide information on any disc quotas that may have been imposed upon a user.

There are two individual possible quotas that may be imposed, usually if one is, both will be. A limit can be set on the amount of space a user can occupy, and there may be a limit on the number of files (inodes) he can own.

Quota provides information on the quotas that have been set by the system administrators, in each of these areas, and current usage.

There are four numbers for each limit, the current usage, soft limit (quota), hard limit, and number of remaining login warnings. The soft limit is the number of 1K blocks (or files) that the user is expected to remain below. Each time the user's usage goes past this limit, he will be warned. The hard limit cannot be exceeded. If a user's usage reaches this number, further requests for space (or attempts to create a file) will fail with an EDQUOT error, and the first time this occurs, a message will be written to the user's terminal. Only one message will be output, until space occupied is reduced below the limit, and reaches it again, in order to avoid continual noise from those programs that ignore write errors.

Whenever a user logs in with a usage greater than his soft limit, he will be warned, and his login warning count decremented. When he logs in under quota, the counter is reset to its maximum value (which is a system configuration parameter, that is typically 3). If the warning count should ever reach zero (caused by three successive logins over quota), the particular limit that has been exceeded will be treated as if the hard limit has been reached, and no more resources will be allocated to the user. The **only** way to reset this condition is to reduce usage below quota, then log in again.

1.1. Surviving when quota limit is reached

In most cases, the only way to recover from over quota conditions, is to abort whatever activity was in progress on the filesystem that has reached its limit, remove sufficient files to bring the limit back below quota, and retry the failed program.

However, if you are in the editor and a write fails because of an over quota situation, that is not a suitable course of action, as it is most likely that initially attempting to write the file will have truncated its previous contents, so should the editor be aborted without correctly writing the file not only will the recent changes be lost, but possibly much, or even all, of the data that previously existed.

There are several possible safe exits for a user caught in this situation. He may use the editor ! shell escape command to examine his file space, and remove surplus files. Alternatively, using

* UNIX is a trademark of Bell Laboratories.

csh, he may suspend the editor, remove some files, then resume it. A third possibility, is to write the file to some other filesystem (perhaps to a file on /tmp) where the user's quota has not been exceeded. Then after rectifying the quota situation, the file can be moved back to the filesystem it belongs on.

2. Administering the quota system

To set up and establish the disc quota system, there are several steps necessary to be performed by the system administrator.

First, the system must be configured to include the disc quota sub-system. This is done by including the line:

options QUOTA

in the system configuration file, then running *config*(8) followed by a system configuration*.

Second, a decision as to what filesystems need to have quotas applied needs to be made. Usually, only filesystems that house users' home directories, or other user files, will need to be subjected to the quota system, though it may also prove useful to also include **/usr**. If possible, **/tmp** should usually be free of quotas.

Having decided on which filesystems quotas need to be set upon, the administrator should then allocate the available space amongst the competing needs. How this should be done is (way) beyond the scope of this document.

Then, the *edquota*(8) command can be used to actually set the limits desired upon each user. Where a number of users are to be given the same quotas (a common occurrence) the **-p** switch to edquota will allow this to be easily accomplished.

Once the quotas are set, ready to operate, the system must be informed to enforce quotas on the desired filesystems. This is accomplished with the *quotaon*(8) command. *Quotaon* will either enable quotas for a particular filesystem, or with the **-a** switch, will enable quotas for each filesystem indicated in **/etc/fstab** as using quotas. See *fstab*(5) for details. Most sites using the quota system, will include the line

/etc/quotaon -a

in **/etc/rc.local**.

Should quotas need to be disabled, the *quotaoff*(8) command will do that, however, should the filesystem be about to be dismounted, the *umount*(8) command will disable quotas immediately before the filesystem is unmounted. This is actually an effect of the *umount*(2) system call, and it guarantees that the quota system will not be disabled if the umount would fail because the filesystem is not idle.

Periodically (certainly after each reboot, and when quotas are first enabled for a filesystem), the records retained in the quota file should be checked for consistency with the actual number of blocks and files allocated to the user. The *quotachk*(8) command can be used to accomplish this. It is not necessary to dismount the filesystem, or disable the quota system to run this command, though on active filesystems inaccurate results may occur. This does no real harm in most cases, another run of *quotachk* when the filesystem is idle will certainly correct any inaccuracy.

The super-user may use the *quota*(1) command to examine the usage and quotas of any user, and the *repquota*(8) command may be used to check the usages and limits for all users on a filesystem.

---

* See also the document "Building 4.2BSD UNIX Systems with Config".

3. Some implementation detail.

Disc quota usage and information is stored in a file on the filesystem that the quotas are to be applied to. Conventionally, this file is **quotas** in the root of the filesystem. While this name is not known to the system in any way, several of the user level utilities "know" it, and choosing any other name would not be wise.

The data in the file comprises an array of structures, indexed by uid, one structure for each user on the system (whether the user has a quota on this filesystem or not). If the uid space is sparse, then the file may have holes in it, which would be lost by copying, so it is best to avoid this.

The system is informed of the existence of the quota file by the *setquota*(2) system call. It then reads the quota entries for each user currently active, then for any files open owned by users who are not currently active. Each subsequent open of a file on the filesystem, will be accompanied by a pairing with its quota information. In most cases this information will be retained in core, either because the user who owns the file is running some process, because other files are open owned by the same user, or because some file (perhaps this one) was recently accessed. In memory, the quota information is kept hashed by user-id and filesystem, and retained in an LRU chain so recently released data can be easily reclaimed. Information about those users whose last process has recently terminated is also retained in this way.

Each time a block is accessed or released, and each time an inode is allocated or freed, the quota system gets told about it, and in the case of allocations, gets the opportunity to object.

Measurements have shown that the quota code uses a very small percentage of the system cpu time consumed in writing a new block to disc.

4. Acknowledgments

The current disc quota system is loosely based upon a very early scheme implemented at the University of New South Wales, and Sydney University in the mid 70's. That system implemented a single combined limit for both files and blocks on all filesystems.

A later system was implemented at the University of Melbourne by the author, but was not kept highly accurately, eg: chown's (etc) did not affect quotas, nor did i/o to a file other than one owned by the instigator.

The current system has been running (with only minor modifications) since January 82 at Melbourne. It is actually just a small part of a much broader resource control scheme, which is capable of controlling almost anything that is usually uncontrolled in unix. The rest of this is, as yet, still in a state where it is far too subject to change to be considered for distribution.

For the 4.2BSD release, much work has been done to clean up and sanely incorporate the quota code by Sam Leffler and Kirk McKusick at The University of California at Berkeley.

SENDMAIL

INSTALLATION AND OPERATION GUIDE

Eric Allman
Britton-Lee, Inc.

Version 4.2

TABLE OF CONTENTS

SENDMAIL

INSTALLATION AND OPERATION GUIDE

Eric Allman
Britton-Lee, Inc.

Version 4.2

Sendmail implements a general purpose internetwork mail routing facility under the UNIX* operating system. It is not tied to any one transport protocol – its function may be likened to a crossbar switch, relaying messages from one domain into another. In the process, it can do a limited amount of message header editing to put the message into a format that is appropriate for the receiving domain. All of this is done under the control of a configuration file.

Due to the requirements of flexibility for *sendmail*, the configuration file can seem somewhat unapproachable. However, there are only a few basic configurations for most sites, for which standard configuration files have been supplied. Most other configurations can be built by adjusting an existing configuration files incrementally.

Although *sendmail* is intended to run without the need for monitoring, it has a number of features that may be used to monitor or adjust the operation under unusual circumstances. These features are described.

Section one describes how to do a basic *sendmail* installation. Section two explains the day-to-day information you should know to maintain your mail system. If you have a relatively normal site, these two sections should contain sufficient information for you to install *sendmail* and keep it happy. Section three describes some parameters that may be safely tweaked. Section four has information regarding the command line arguments. Section five contains the nitty-gritty information about the configuration file. This section is for masochists and people who must write their own configuration file. The appendixes give a brief but detailed explanation of a number of features not described in the rest of the paper.

The references in this paper are actually found in the companion paper *Sendmail – An Internetwork Mail Router*. This other paper should be read before this manual to gain a basic understanding of how the pieces fit together.

1. BASIC INSTALLATION

There are two basic steps to installing sendmail. The hard part is to build the configuration table. This is a file that sendmail reads when it starts up that describes the mailers it knows about, how to parse addresses, how to rewrite the message header, and the settings of various options. Although the configuration table is quite complex, a configuration can usually be built by adjusting an existing off-the-shelf configuration. The second part is actually doing the installation, i.e., creating the necessary files, etc.

The remainder of this section will describe the installation of sendmail assuming you can use one of the existing configurations and that the standard installation parameters are acceptable. All pathnames and examples are given from the root of the *sendmail* subtree.

---

*UNIX is a trademark of Bell Laboratories.

1.1. Off-The-Shelf Configurations

The configuration files are all in the subdirectory *cf* of the sendmail directory. The ones used at Berkeley are in *m4*(1) format; files with names ending ".m4" are *m4* include files, while files with names ending ".mc" are the master files. Files with names ending ".cf" are the *m4* processed versions of the corresponding ".mc" file.

Two off the shelf configuration files are supplied to handle the basic cases: *cf/arpaproto.cf* for Arpanet (TCP) sites and *cf/uucpproto.cf* for UUCP sites. These are *not* in *m4* format. The file you need should be copied to a file with the same name as your system, e.g.,

```
cp uucpproto.cf ucsfcgl.cf
```

This file is now ready for installation as */usr/lib/sendmail.cf*.

1.2. Installation Using the Makefile

A makefile exists in the root of the *sendmail* directory that will do all of these steps for a 4.2BSD system. It may have to be slightly tailored for use on other systems.

Before using this makefile, you should already have created your configuration file and left it in the file "cf/*system*.cf" where *system* is the name of your system (i.e., what is returned by *hostname*(1)). If you do not have *hostname* you can use the declaration "HOST=*system*" on the *make*(1) command line. You should also examine the file *md/config.m4* and change the *m4* macros there to reflect any libraries and compilation flags you may need.

The basic installation procedure is to type:

```
make
make install
```

in the root directory of the *sendmail* distribution. This will make all binaries and install them in the standard places. The second *make* command must be executed as the superuser (root).

1.3. Installation by Hand

Along with building a configuration file, you will have to install the *sendmail* startup into your UNIX system. If you are doing this installation in conjunction with a regular Berkeley UNIX install, these steps will already be complete. Many of these steps will have to be executed as the superuser (root).

1.3.1. lib/libsys.a

The library in lib/libsys.a contains some routines that should in some sense be part of the system library. These are the system logging routines and the new directory access routines (if required). If you are not running the new 4.2BSD directory code and do not have the compatibility routines installed in your system library, you should execute the commands:

```
cd lib
make ndir
```

This will compile and install the 4.2 compatibility routines in the library. You should then type:

```
cd lib      # if required
make
```

This will recompile and fill the library.

1.3.2. /usr/lib/sendmail

The binary for sendmail is located in /usr/lib. There is a version available in the source directory that is probably inadequate for your system. You should plan on recompiling and installing the entire system:

```
cd src
rm -f *.o
make
cp sendmail /usr/lib
```

1.3.3. /usr/lib/sendmail.cf

The configuration file that you created earlier should be installed in /usr/lib/sendmail.cf:

```
cp cf/system.cf /usr/lib/sendmail.cf
```

1.3.4. /usr/ucb/newaliases

If you are running delivermail, it is critical that the *newaliases* command be replaced. This can just be a link to *sendmail*:

```
rm -f /usr/ucb/newaliases
ln /usr/lib/sendmail /usr/ucb/newaliases
```

1.3.5. /usr/lib/sendmail.cf

The configuration file must be installed in /usr/lib. This is described above.

1.3.6. /usr/spool/mqueue

The directory */usr/spool/mqueue* should be created to hold the mail queue. This directory should be mode 777 unless *sendmail* is run setuid, when *mqueue* should be owned by the sendmail owner and mode 755.

1.3.7. /usr/lib/aliases*

The system aliases are held in three files. The file "/usr/lib/aliases" is the master copy. A sample is given in "lib/aliases" which includes some aliases which *must* be defined:

```
cp lib/aliases /usr/lib/aliases
```

You should extend this file with any aliases that are apropos to your system.

Normally *sendmail* looks at a version of these files maintained by the *dbm*(3) routines. These are stored in "/usr/lib/aliases.dir" and "/usr/lib/aliases.pag." These can initially be created as empty files, but they will have to be initialized promptly. These should be mode 666 if you are running a reasonably relaxed system:

```
cp /dev/null /usr/lib/aliases.dir
cp /dev/null /usr/lib/aliases.pag
chmod 666 /usr/lib/aliases.*
newaliases
```

1.3.8. /usr/lib/sendmail.fc

If you intend to install the frozen version of the configuration file (for quick startup) you should create the file /usr/lib/sendmail.fc and initialize it. This step may be safely skipped.

```
cp /dev/null /usr/lib/sendmail.fc
/usr/lib/sendmail -bz
```

1.3.9. /etc/rc

It will be necessary to start up the sendmail daemon when your system reboots. This daemon performs two functions: it listens on the SMTP socket for connections (to receive mail from a remote system) and it processes the queue periodically to insure that mail gets delivered when hosts come up.

Add the following lines to "/etc/rc" (or "/etc/rc.local" as appropriate) in the area where it is starting up the daemons:

```
if [ -f /usr/lib/sendmail ]; then
        (cd /usr/spool/mqueue; rm -f [lnx]f*)
        /usr/lib/sendmail -bd -q30m &
        echo -n ' sendmail' >/dev/console
fi
```

The "cd" and "rm" commands insure that all lock files have been removed; extraneous lock files may be left around if the system goes down in the middle of processing a message. The line that actually invokes *sendmail* has two flags: "-bd" causes it to listen on the SMTP port, and "-q30m" causes it to run the queue every half hour.

If you are not running a version of UNIX that supports Berkeley TCP/IP, do not include the **-bd** flag.

1.3.10. /usr/lib/sendmail.hf

This is the help file used by the SMTP **HELP** command. It should be copied from "lib/sendmail.hf":

```
cp lib/sendmail.hf /usr/lib
```

1.3.11. /usr/lib/sendmail.st

If you wish to collect statistics about your mail traffic, you should create the file "/usr/lib/sendmail.st":

```
cp /dev/null /usr/lib/sendmail.st
chmod 666 /usr/lib/sendmail.st
```

This file does not grow. It is printed with the program "aux/mailstats."

1.3.12. /etc/syslog

You may want to run the *syslog* program (to collect log information about sendmail). This program normally resides in */etc/syslog*, with support files */etc/syslog.conf* and */etc/syslog.pid*. The program is located in the *aux* subdirectory of the *sendmail* distribution. The file */etc/syslog.conf* describes the file(s) that sendmail will log in. For a complete description of syslog, see the manual page for *syslog*(8) (located in *sendmail/doc* on the distribution).

1.3.13. /usr/ucb/newaliases

If *sendmail* is invoked as "newaliases," it will simulate the **-bi** flag (i.e., will rebuild the alias database; see below). This should be a link to /usr/lib/sendmail.

1.3.14. /usr/ucb/mailq

If *sendmail* is invoked as "mailq," it will simulate the **-bp** flag (i.e., *sendmail* will print the contents of the mail queue; see below). This should be a link to /usr/lib/sendmail.

2. NORMAL OPERATIONS

2.1. Quick Configuration Startup

A fast version of the configuration file may be set up by using the **-bz** flag:

/usr/lib/sendmail -bz

This creates the file */usr/lib/sendmail.fc* ("frozen configuration"). This file is an image of *sendmail*'s data space after reading in the configuration file. If this file exists, it is used instead of */usr/lib/sendmail.cf* *sendmail.fc* must be rebuilt manually every time *sendmail.cf* is changed.

The frozen configuration file will be ignored if a **-C** flag is specified or if sendmail detects that it is out of date. However, the heuristics are not strong so this should not be trusted.

2.2. The System Log

The system log is supported by the *syslog*(8) program.

2.2.1. Format

Each line in the system log consists of a timestamp, the name of the machine that generated it (for logging from several machines over the ethernet), the word "sendmail:", and a message.

2.2.2. Levels

If you have *syslog*(8) or an equivalent installed, you will be able to do logging. There is a large amount of information that can be logged. The log is arranged as a succession of levels. At the lowest level only extremely strange situations are logged. At the highest level, even the most mundane and uninteresting events are recorded for posterity. As a convention, log levels under ten are considered "useful;" log levels above ten are usually for debugging purposes.

A complete description of the log levels is given in section 4.3.

2.3. The Mail Queue

The mail queue should be processed transparently. However, you may find that manual intervention is sometimes necessary. For example, if a major host is down for a period of time the queue may become clogged. Although sendmail ought to recover gracefully when the host comes up, you may find performance unacceptably bad in the meantime.

2.3.1. Printing the queue

The contents of the queue can be printed using the *mailq* command (or by specifying the **-bp** flag to sendmail):

mailq

This will produce a listing of the queue id's, the size of the message, the date the message entered the queue, and the sender and recipients.

2.3.2. Format of queue files

All queue files have the form *xfAA99999* where *AA99999* is the *id* for this file and the *x* is a type. The types are:

d The data file. The message body (excluding the header) is kept in this file.

l The lock file. If this file exists, the job is currently being processed, and a queue run will not process the file. For that reason, an extraneous **lf** file can cause a job

to apparently disappear (it will not even time out!).

n This file is created when an id is being created. It is a separate file to insure that no mail can ever be destroyed due to a race condition. It should exist for no more than a few milliseconds at any given time.

q The queue control file. This file contains the information necessary to process the job.

t A temporary file. These are an image of the **qf** file when it is being rebuilt. It should be renamed to a **qf** file very quickly.

x A transcript file, existing during the life of a session showing everything that happens during that session.

The **qf** file is structured as a series of lines each beginning with a code letter. The lines are as follows:

D The name of the data file. There may only be one of these lines.

H A header definition. There may be any number of these lines. The order is important: they represent the order in the final message. These use the same syntax as header definitions in the configuration file.

R A recipient address. This will normally be completely aliased, but is actually realiased when the job is processed. There will be one line for each recipient.

S The sender address. There may only be one of these lines.

T The job creation time. This is used to compute when to time out the job.

P The current message priority. This is used to order the queue. Higher numbers mean lower priorities. The priority increases as the message sits in the queue. The initial priority depends on the message class and the size of the message.

M A message. This line is printed by the *mailq* command, and is generally used to store status information. It can contain any text.

As an example, the following is a queue file sent to "mckusick@calder" and "wnj":

```
DdfA13557
Seric
T404261372
P132
Rmckusick@calder
Rwnj
H?D?date: 23-Oct-82 15:49:32-PDT (Sat)
H?F?from: eric (Eric Allman)
H?x?full-name: Eric Allman
Hsubject: this is an example message
Hmessage-id: <8209232249.13557@UCBARPA.BERKELEY.ARPA>
Hreceived: by UCBARPA.BERKELEY.ARPA (3.227 [10/22/82])
        id A13557; 23-Oct-82 15:49:32-PDT (Sat)
Hphone: (415) 548-3211
HTo: mckusick@calder, wnj
```

This shows the name of the data file, the person who sent the message, the submission time (in seconds since January 1, 1970), the message priority, the message class, the recipients, and the headers for the message.

2.3.3. Forcing the queue

Sendmail should run the queue automatically at intervals. The algorithm is to read and sort the queue, and then to attempt to process all jobs in order. When it attempts to run the job, *sendmail* first checks to see if the job is locked. If so, it ignores

the job.

There is no attempt to insure that only one queue processor exists at any time, since there is no guarantee that a job cannot take forever to process. Due to the locking algorithm, it is impossible for one job to freeze the queue. However, an uncooperative recipient host or a program recipient that never returns can accumulate many processes in your system. Unfortunately, there is no way to resolve this without violating the protocol.

In some cases, you may find that a major host going down for a couple of days may create a prohibitively large queue. This will result in *sendmail* spending an inordinate amount of time sorting the queue. This situation can be fixed by moving the queue to a temporary place and creating a new queue. The old queue can be run later when the offending host returns to service.

To do this, it is acceptable to move the entire queue directory:

```
cd /usr/spool
mv mqueue omqueue; mkdir mqueue; chmod 777 mqueue
```

You should then kill the existing daemon (since it will still be processing in the old queue directory) and create a new daemon.

To run the old mail queue, run the following command:

```
/usr/lib/sendmail -oQ/usr/spool/omqueue -q
```

The **-oQ** flag specifies an alternate queue directory and the **-q** flag says to just run every job in the queue. If you have a tendency toward voyeurism, you can use the **-v** flag to watch what is going on.

When the queue is finally emptied, you can remove the directory:

```
rmdir /usr/spool/omqueue
```

2.4. The Alias Database

The alias database exists in two forms. One is a text form, maintained in the file */usr/lib/aliases*. The aliases are of the form

```
name: name1, name2, ...
```

Only local names may be aliased; e.g.,

```
eric@mit-xx: eric@berkeley
```

will not have the desired effect. Aliases may be continued by starting any continuation lines with a space or a tab. Blank lines and lines beginning with a sharp sign ("#") are comments.

The second form is processed by the *dbm*(3) library. This form is in the files */usr/lib/aliases.dir* and */usr/lib/aliases.pag*. This is the form that *sendmail* actually uses to resolve aliases. This technique is used to improve performance.

2.4.1. Rebuilding the alias database

The DBM version of the database may be rebuilt explicitly by executing the command

```
newaliases
```

This is equivalent to giving *sendmail* the **-bi** flag:

```
/usr/lib/sendmail -bi
```

If the "D" option is specified in the configuration, *sendmail* will rebuild the alias database automatically if possible when it is out of date. The conditions under which it will do this are:

(1) The DBM version of the database is mode 666. -or-

(2) *Sendmail* is running setuid to root.

Auto-rebuild can be dangerous on heavily loaded machines with large alias files; if it might take more than five minutes to rebuild the database, there is a chance that several processes will start the rebuild process simultaneously.

2.4.2. Potential problems

There are a number of problems that can occur with the alias database. They all result from a *sendmail* process accessing the DBM version while it is only partially built. This can happen under two circumstances: One process accesses the database while another process is rebuilding it, or the process rebuilding the database dies (due to being killed or a system crash) before completing the rebuild.

Sendmail has two techniques to try to relieve these problems. First, it ignores interrupts while rebuilding the database; this avoids the problem of someone aborting the process leaving a partially rebuilt database. Second, at the end of the rebuild it adds an alias of the form

@: @

(which is not normally legal). Before sendmail will access the database, it checks to insure that this entry exists[1]. It will wait up to five minutes for this entry to appear, at which point it will force a rebuild itself[2].

2.4.3. List owners

If an error occurs on sending to a certain address, say "*x*", *sendmail* will look for an alias of the form "owner-*x*" to receive the errors. This is typically useful for a mailing list where the submitter of the list has no control over the maintanence of the list itself; in this case the list maintainer would be the owner of the list. For example:

```
unix-wizards: eric@ucbarpa, wnj@monet, nosuchuser,
        sam@matisse
owner-unix-wizards: eric@ucbarpa
```

would cause "eric@ucbarpa" to get the error that will occur when someone sends to unix-wizards due to the inclusion of "nosuchuser" on the list.

2.5. Per-User Forwarding (.forward Files)

As an alternative to the alias database, any user may put a file with the name ".forward" in his or her home directory. If this file exists, *sendmail* redirects mail for that user to the list of addresses listed in the .forward file. For example, if the home directory for user "mckusick" has a .forward file with contents:

```
mckusick@ernie
kirk@calder
```

then any mail arriving for "mckusick" will be redirected to the specified accounts.

2.6. Special Header Lines

Several header lines have special interpretations defined by the configuration file. Others have interpretations built into *sendmail* that cannot be changed without changing the code. These builtins are described here.

---

[1]The "a" option is required in the configuration for this action to occur. This should normally be specified unless you are running *delivermail* in parallel with *sendmail*.

[2]Note: the "D" option must be specified in the configuration file for this operation to occur.

2.6.1. Return-Receipt-To:

If this header is sent, a message will be sent to any specified addresses when the final delivery is complete. if the mailer has the **l** flag (local delivery) set in the mailer descriptor.

2.6.2. Errors-To:

If errors occur anywhere during processing, this header will cause error messages to go to the listed addresses rather than to the sender. This is intended for mailing lists.

2.6.3. Apparently-To:

If a message comes in with no recipients listed in the message (in a To:, Cc:, or Bcc: line) then *sendmail* will add an "Apparently-To:" header line for any recipients it is aware of. This is not put in as a standard recipient line to warn any recipients that the list is not complete.

At least one recipient line is required under RFC 822.

3. ARGUMENTS

The complete list of arguments to *sendmail* is described in detail in Appendix A. Some important arguments are described here.

3.1. Queue Interval

The amount of time between forking a process to run through the queue is defined by the **–q** flag. If you run in mode **f** or **a** this can be relatively large, since it will only be relevant when a host that was down comes back up. If you run in **q** mode it should be relatively short, since it defines the maximum amount of time that a message may sit in the queue.

3.2. Daemon Mode

If you allow incoming mail over an IPC connection, you should have a daemon running. This should be set by your */etc/rc* file using the **–bd** flag. The **–bd** flag and the **–q** flag may be combined in one call:

```
/usr/lib/sendmail –bd –q30m
```

3.3. Forcing the Queue

In some cases you may find that the queue has gotten clogged for some reason. You can force a queue run using the **–q** flag (with no value). It is entertaining to use the **–v** flag (verbose) when this is done to watch what happens:

```
/usr/lib/sendmail –q –v
```

3.4. Debugging

There are a fairly large number of debug flags built into *sendmail*. Each debug flag has a number and a level, where higher levels means to print out more information. The convention is that levels greater than nine are "absurd," i.e., they print out so much information that you wouldn't normally want to see them except for debugging that particular piece of code. Debug flags are set using the **–d** option; the syntax is:

```
debug-flag:    –d debug-list
debug-list:    debug-option [ , debug-option ]
debug-option:  debug-range [ . debug-level ]
debug-range:   integer | integer – integer
debug-level:   integer
```

where spaces are for reading ease only. For example,

–d12	Set flag 12 to level 1
–d12.3	Set flag 12 to level 3
–d3-17	Set flags 3 through 17 to level 1
–d3-17.4	Set flags 3 through 17 to level 4

For a complete list of the available debug flags you will have to look at the code (they are too dynamic to keep this documentation up to date).

3.5. Trying a Different Configuration File

An alternative configuration file can be specified using the **–C** flag; for example,

/usr/lib/sendmail –Ctest.cf

uses the configuration file *test.cf* instead of the default */usr/lib/sendmail.cf*. If the **–C** flag has no value it defaults to *sendmail.cf* in the current directory.

3.6. Changing the Values of Options

Options can be overridden using the **–o** flag. For example,

/usr/lib/sendmail –oT2m

sets the **T** (timeout) option to two minutes for this run only.

4. TUNING

There are a number of configuration parameters you may want to change, depending on the requirements of your site. Most of these are set using an option in the configuration file. For example, the line "OT3d" sets option "T" to the value "3d" (three days).

4.1. Timeouts

All time intervals are set using a scaled syntax. For example, "10m" represents ten minutes, whereas "2h30m" represents two and a half hours. The full set of scales is:

s	seconds
m	minutes
h	hours
d	days
w	weeks

4.1.1. Queue interval

The argument to the **–q** flag specifies how often a subdaemon will run the queue. This is typically set to between five minutes and one half hour.

4.1.2. Read timeouts

It is possible to time out when reading the standard input or when reading from a remote SMTP server. Technically, this is not acceptable within the published protocols. However, it might be appropriate to set it to something large in certain environments (such as an hour). This will reduce the chance of large numbers of idle daemons piling up on your system. This timeout is set using the **r** option in the configuration file.

4.1.3. Message timeouts

After sitting in the queue for a few days, a message will time out. This is to insure that at least the sender is aware of the inability to send a message. The timeout is typically set to three days. This timeout is set using the **T** option in the configuration file.

The time of submission is set in the queue, rather than the amount of time left until timeout. As a result, you can flush messages that have been hanging for a short period by running the queue with a short message timeout. For example,

```
/usr/lib/sendmail -oT1d -q
```

will run the queue and flush anything that is one day old.

4.2. Delivery Mode

There are a number of delivery modes that *sendmail* can operate in, set by the "d" configuration option. These modes specify how quickly mail will be delivered. Legal modes are:

- i deliver interactively (synchronously)
- b deliver in background (asynchronously)
- q queue only (don't deliver)

There are tradeoffs. Mode "i" passes the maximum amount of information to the sender, but is hardly ever necessary. Mode "q" puts the minimum load on your machine, but means that delivery may be delayed for up to the queue interval. Mode "b" is probably a good compromise. However, this mode can cause large numbers of processes if you have a mailer that takes a long time to deliver a message.

4.3. Log Level

The level of logging can be set for sendmail. The default using a standard configuration table is level 9. The levels are as follows:

- 0 No logging.
- 1 Major problems only.
- 2 Message collections and failed deliveries.
- 3 Successful deliveries.
- 4 Messages being defered (due to a host being down, etc.).
- 5 Normal message queueups.
- 6 Unusual but benign incidents, e.g., trying to process a locked queue file.
- 9 Log internal queue id to external message id mappings. This can be useful for tracing a message as it travels between several hosts.
- 12 Several messages that are basically only of interest when debugging.
- 16 Verbose information regarding the queue.

4.4. File Modes

There are a number of files that may have a number of modes. The modes depend on what functionality you want and the level of security you require.

4.4.1. To suid or not to suid?

Sendmail can safely be made setuid to root. At the point where it is about to *exec*(2) a mailer, it checks to see if the userid is zero; if so, it resets the userid and groupid to a default (set by the **u** and **g** options). (This can be overridden by setting the **S** flag to the mailer for mailers that are trusted and must be called as root.)

However, this will cause mail processing to be accounted (using *sa*(8)) to root rather than to the user sending the mail.

4.4.2. Temporary file modes

The mode of all temporary files that *sendmail* creates is determined by the "F" option. Reasonable values for this option are 0600 and 0644. If the more permissive mode is selected, it will not be necessary to run *sendmail* as root at all (even when running the queue).

4.4.3. Should my alias database be writable?

At Berkeley we have the alias database (/usr/lib/aliases*) mode 666. There are some dangers inherent in this approach: any user can add him-/her-self to any list, or can "steal" any other user's mail. However, we have found users to be basically trustworthy, and the cost of having a read-only database greater than the expense of finding and eradicating the rare nasty person.

The database that *sendmail* actually used is represented by the two files *aliases.dir* and *aliases.pag* (both in /usr/lib). The mode on these files should match the mode on /usr/lib/aliases. If *aliases* is writable and the DBM files (*aliases.dir* and *aliases.pag*) are not, users will be unable to reflect their desired changes through to the actual database. However, if *aliases* is read-only and the DBM files are writable, a slightly sophisticated user can arrange to steal mail anyway.

If your DBM files are not writable by the world or you do not have auto-rebuild enabled (with the "D" option), then you must be careful to reconstruct the alias database each time you change the text version:

```
newaliases
```

If this step is ignored or forgotten any intended changes will also be ignored or forgotten.

5. THE WHOLE SCOOP ON THE CONFIGURATION FILE

This section describes the configuration file in detail, including hints on how to write one of your own if you have to.

There is one point that should be made clear immediately: the syntax of the configuration file is designed to be reasonably easy to parse, since this is done every time *sendmail* starts up, rather than easy for a human to read or write. On the "future project" list is a configuration-file compiler.

An overview of the configuration file is given first, followed by details of the semantics.

5.1. The Syntax

The configuration file is organized as a series of lines, each of which begins with a single character defining the semantics for the rest of the line. Lines beginning with a space or a tab are continuation lines (although the semantics are not well defined in many places). Blank lines and lines beginning with a sharp symbol ('#') are comments.

5.1.1. R and S – rewriting rules

The core of address parsing are the rewriting rules. These are an ordered production system. *Sendmail* scans through the set of rewriting rules looking for a match on the left hand side (LHS) of the rule. When a rule matches, the address is replaced by the right hand side (RHS) of the rule.

There are several sets of rewriting rules. Some of the rewriting sets are used internally and must have specific semantics. Other rewriting sets do not have specifically assigned semantics, and may be referenced by the mailer definitions or by other

rewriting sets.

The syntax of these two commands are:

S*n*

Sets the current ruleset being collected to *n*. If you begin a ruleset more than once it deletes the old definition.

R*lhs rhs comments*

The fields must be separated by at least one tab character; there may be embedded spaces in the fields. The *lhs* is a pattern that is applied to the input. If it matches, the input is rewritten to the *rhs*. The *comments* are ignored.

5.1.2. D – define macro

Macros are named with a single character. These may be selected from the entire ASCII set, but user-defined macros should be selected from the set of upper case letters only. Lower case letters and special symbols are used internally.

The syntax for macro definitions is:

D*x val*

where *x* is the name of the macro and *val* is the value it should have. Macros can be interpolated in most places using the escape sequence $*x*.

5.1.3. C and F – define classes

Classes of words may be defined to match on the left hand side of rewriting rules. For example a class of all local names for this site might be created so that attempts to send to oneself can be eliminated. These can either be defined directly in the configuration file or read in from another file. Classes may be given names from the set of upper case letters. Lower case letters and special characters are reserved for system use.

The syntax is:

C*c word1 word2...*
F*c file* [*format*]

The first form defines the class *c* to match any of the named words. It is permissible to split them among multiple lines; for example, the two forms:

CHmonet ucbmonet

and

CHmonet
CHucbmonet

are equivalent. The second form reads the elements of the class *c* from the named *file*; the *format* is a *scanf*(3) pattern that should produce a single string.

5.1.4. M – define mailer

Programs and interfaces to mailers are defined in this line. The format is:

M*name*, {*field*=*value*}*

where *name* is the name of the mailer (used internally only) and the "field=name" pairs define attributes of the mailer. Fields are:

Path	The pathname of the mailer
Flags	Special flags for this mailer
Sender	A rewriting set for sender addresses
Recipient	A rewriting set for recipient addresses
Argv	An argument vector to pass to this mailer
Eol	The end-of-line string for this mailer
Maxsize	The maximum message length to this mailer

Only the first character of the field name is checked.

5.1.5. H – define header

The format of the header lines that sendmail inserts into the message are defined by the **H** line. The syntax of this line is:

H[?*mflags*?]*hname*: *htemplate*

Continuation lines in this spec are reflected directly into the outgoing message. The *htemplate* is macro expanded before insertion into the message. If the *mflags* (surrounded by question marks) are specified, at least one of the specified flags must be stated in the mailer definition for this header to be automatically output. If one of these headers is in the input it is reflected to the output regardless of these flags.

Some headers have special semantics that will be described below.

5.1.6. O – set option

There are a number of "random" options that can be set from a configuration file. Options are represented by single characters. The syntax of this line is:

O*o value*

This sets option *o* to be *value*. Depending on the option, *value* may be a string, an integer, a boolean (with legal values "t", "T", "f", or "F"; the default is TRUE), or a time interval.

5.1.7. T – define trusted users

Trusted users are those users who are permitted to override the sender address using the **–f** flag. These typically are "root," "uucp," and "network," but on some users it may be convenient to extend this list to include other users, perhaps to support a separate UUCP login for each host. The syntax of this line is:

T*user1 user2...*

There may be more than one of these lines.

5.1.8. P – precedence definitions

Values for the "Precedence:" field may be defined using the **P** control line. The syntax of this field is:

P*name*=*num*

When the *name* is found in a "Precedence:" field, the message class is set to *num*. Higher numbers mean higher precedence. Numbers less than zero have the special property that error messages will not be returned. The default precedence is zero. For example, our list of precedences is:

```
Pfirst-class=0
Pspecial-delivery=100
Pjunk=-100
```

5.2. The Semantics

This section describes the semantics of the configuration file.

5.2.1. Special macros, conditionals

Macros are interpolated using the construct **$*x***, where *x* is the name of the macro to be interpolated. In particular, lower case letters are reserved to have special semantics, used to pass information in or out of sendmail, and some special characters are reserved to provide conditionals, etc.

The following macros *must* be defined to transmit information into *sendmail:*

e The SMTP entry message
j The "official" domain name for this site
l The format of the UNIX from line
n The name of the daemon (for error messages)
o The set of "operators" in addresses
q default format of sender address

The **$e** macro is printed out when SMTP starts up. The first word must be the **$j** macro. The **$j** macro should be in RFC821 format. The **$l** and **$n** macros can be considered constants except under terribly unusual circumstances. The **$o** macro consists of a list of characters which will be considered tokens and which will separate tokens when doing parsing. For example, if "r" were in the **$o** macro, then the input "address" would be scanned as three tokens: "add," "r," and "ess." Finally, the **$q** macro specifies how an address should appear in a message when it is defaulted. For example, on our system these definitions are:

```
De$j Sendmail $v ready at $b
DnMAILER-DAEMON
DlFrom $g  $d
Do.:%@!^=/
Dq$g$?x ($x)$.
Dj$H.$D
```

An acceptable alternative for the **$q** macro is "$?x$x $.<$g>". These correspond to the following two formats:

```
eric@Berkeley (Eric Allman)
Eric Allman <eric@Berkeley>
```

Some macros are defined by *sendmail* for interpolation into argv's for mailers or for other contexts. These macros are:

a The origination date in Arpanet format
b The current date in Arpanet format
c The hop count
d The date in UNIX (ctime) format
f The sender (from) address
g The sender address relative to the recipient
h The recipient host
i The queue id
p Sendmail's pid
r Protocol used
s Sender's host name
t A numeric representation of the current time
u The recipient user
v The version number of sendmail
w The hostname of this site
x The full name of the sender
y The id of the sender's tty
z The home directory of the recipient

There are three types of dates that can be used. The **$a** and **$b** macros are in Arpanet format; **$a** is the time as extracted from the "Date:" line of the message (if there was one), and **$b** is the current date and time (used for postmarks). If no "Date:" line is found in the incoming message, **$a** is set to the current time also. The **$d** macro is equivalent to the **$a** macro in UNIX (ctime) format.

The **$f** macro is the id of the sender as originally determined; when mailing to a specific host the **$g** macro is set to the address of the sender *relative to the recipient.* For example, if I send to "bollard@matisse" from the machine "ucbarpa" the **$f** macro will be "eric" and the **$g** macro will be "eric@ucbarpa."

The **$x** macro is set to the full name of the sender. This can be determined in several ways. It can be passed as flag to *sendmail.* The second choice is the value of the "Full-name:" line in the header if it exists, and the third choice is the comment field of a "From:" line. If all of these fail, and if the message is being originated locally, the full name is looked up in the */etc/passwd* file.

When sending, the **$h**, **$u**, and **$z** macros get set to the host, user, and home directory (if local) of the recipient. The first two are set from the **$@** and **$:** part of the rewriting rules, respectively.

The **$p** and **$t** macros are used to create unique strings (e.g., for the "Message-Id:" field). The **$i** macro is set to the queue id on this host; if put into the timestamp line it can be extremely useful for tracking messages. The **$y** macro is set to the id of the terminal of the sender (if known); some systems like to put this in the Unix "From" line. The **$v** macro is set to be the version number of *sendmail*; this is normally put in timestamps and has been proven extremely useful for debugging. The **$w** macro is set to the name of this host if it can be determined. The **$c** field is set to the "hop count," i.e., the number of times this message has been processed. This can be determined by the **–h** flag on the command line or by counting the timestamps in the message.

The **$r** and **$s** fields are set to the protocol used to communicate with sendmail and the sending hostname; these are not supported in the current version.

Conditionals can be specified using the syntax:

$?**x text1 **$| text2 **$.**

This interpolates *text1* if the macro **$x** is set, and *text2* otherwise. The "else" (**$|**) clause may be omitted.

5.2.2. Special classes

The class $=w is set to be the set of all names this host is known by. This can be used to delete local hostnames.

5.2.3. The left hand side

The left hand side of rewriting rules contains a pattern. Normal words are simply matched directly. Metasyntax is introduced using a dollar sign. The metasymbols are:

$*	Match zero or more tokens
$+	Match one or more tokens
$–	Match exactly one token
$=*x*	Match any token in class *x*
$~*x*	Match any token not in class *x*

If any of these match, they are assigned to the symbol $*n* for replacement on the right hand side, where *n* is the index in the LHS. For example, if the LHS:

$–:$+

is applied to the input:

UCBARPA:eric

the rule will match, and the values passed to the RHS will be:

$1	UCBARPA
$2	eric

5.2.4. The right hand side

When the right hand side of a rewriting rule matches, the input is deleted and replaced by the right hand side. Tokens are copied directly from the RHS unless they are begin with a dollar sign. Metasymbols are:

$*n*	Substitute indefinite token *n* from LHS
$>*n*	"Call" ruleset *n*
$#*mailer*	Resolve to *mailer*
$@*host*	Specify *host*
$:*user*	Specify *user*

The $*n* syntax substitutes the corresponding value from a $+, $–, $*, $=, or $~ match on the LHS. It may be used anywhere.

The $>*n* syntax causes the remainder of the line to be substituted as usual and then passed as the argument to ruleset *n*. The final value of ruleset *n* then becomes the substitution for this rule.

The $# syntax should *only* be used in ruleset zero. It causes evaluation of the ruleset to terminate immediately, and signals to sendmail that the address has completely resolved. The complete syntax is:

$#*mailer*$@*host*$:*user*

This specifies the {mailer, host, user} 3-tuple necessary to direct the mailer. If the mailer is local the host part may be omitted. The *mailer* and *host* must be a single word, but the *user* may be multi-part.

A RHS may also be preceeded by a $@ or a $: to control evaluation. A $@ prefix causes the ruleset to return with the remainder of the RHS as the value. A $: prefix causes the rule to terminate immediately, but the ruleset to continue; this can be used to avoid continued application of a rule. The prefix is stripped before continuing.

The $@ and $: prefixes may preceed a $> spec; for example:

```
R$+     $:$>7$1
```

matches anything, passes that to ruleset seven, and continues; the **$:** is necessary to avoid an infinite loop.

5.2.5. Semantics of rewriting rule sets

There are five rewriting sets that have specific semantics. These are related as depicted by figure 2.

Ruleset three should turn the address into "canonical form." This form should have the basic syntax:

local-part@host-domain-spec

If no "@" sign is specified, then the host-domain-spec *may* be appended from the sender address (if the **C** flag is set in the mailer definition corresponding to the *sending* mailer). Ruleset three is applied by sendmail before doing anything with any address.

Ruleset zero is applied after ruleset three to addresses that are going to actually specify recipients. It must resolve to a {*mailer, host, user*} triple. The *mailer* must be defined in the mailer definitions from the configuration file. The *host* is defined into the **$h** macro for use in the argv expansion of the specified mailer.

Rulesets one and two are applied to all sender and recipient addresses respectively. They are applied before any specification in the mailer definition. They must never resolve.

Ruleset four is applied to all addresses in the message. It is typically used to translate internal to external form.

5.2.6. Mailer flags etc.

There are a number of flags that may be associated with each mailer, each identified by a letter of the alphabet. Many of them are assigned semantics internally. These are detailed in Appendix C. Any other flags may be used freely to conditionally assign headers to messages destined for particular mailers.

5.2.7. The "error" mailer

The mailer with the special name "error" can be used to generate a user error. The (optional) host field is a numeric exit status to be returned, and the user field is a message to be printed. For example, the entry:

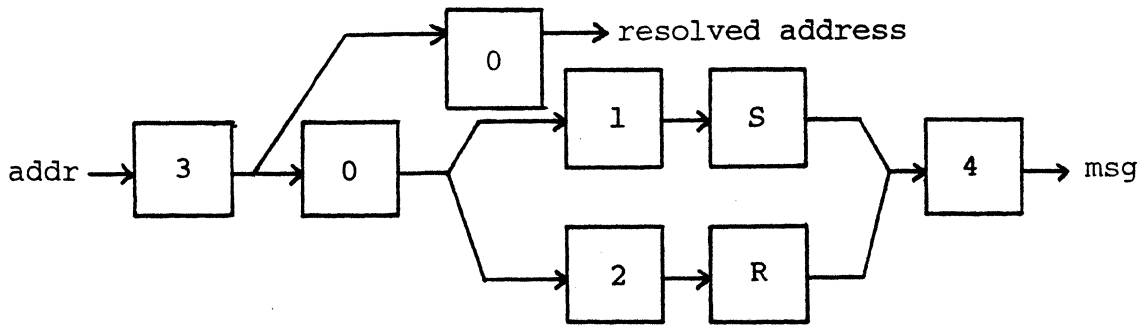


Figure 2 - Rewriting Set Semantics

D — Sender domain addition S — mailer-specific sender rewriting
R — mailer-specific recipient rewriting

$#error$:Host unknown in this domain

on the RHS of a rule will cause the specified error to be generated if the LHS matches. This mailer is only functional in ruleset zero.

5.3. Building a Configuration File From Scratch

Building a configuration table from scratch is an extremely difficult job. Fortunately, it is almost never necessary to do so; nearly every situation that may come up may be resolved by changing an existing table. In any case, it is critical that you understand what it is that you are trying to do and come up with a philosophy for the configuration table. This section is intended to explain what the real purpose of a configuration table is and to give you some ideas for what your philosophy might be.

5.3.1. What you are trying to do

The configuration table has three major purposes. The first and simplest is to set up the environment for *sendmail*. This involves setting the options, defining a few critical macros, etc. Since these are described in other places, we will not go into more detail here.

The second purpose is to rewrite addresses in the message. This should typically be done in two phases. The first phase maps addresses in any format into a canonical form. This should be done in ruleset three. The second phase maps this canonical form into the syntax appropriate for the receiving mailer. *Sendmail* does this in three subphases. Rulesets one and two are applied to all sender and recipient addresses respectively. After this, you may specify per-mailer rulesets for both sender and recipient addresses; this allows mailer-specific customization. Finally, ruleset four is applied to do any default conversion to external form.

The third purpose is to map addresses into the actual set of instructions necessary to get the message delivered. Ruleset zero must resolve to the internal form, which is in turn used as a pointer to a mailer descriptor. The mailer descriptor describes the interface requirements of the mailer.

5.3.2. Philosophy

The particular philosophy you choose will depend heavily on the size and structure of your organization. I will present a few possible philosophies here.

One general point applies to all of these philosophies: it is almost always a mistake to try to do full name resolution. For example, if you are trying to get names of the form "user@host" to the Arpanet, it does not pay to route them to "xyzvax!decvax!ucbvax!c70:user@host" since you then depend on several links not under your control. The best approach to this problem is to simply forward to "xyzvax!user@host" and let xyzvax worry about it from there. In summary, just get the message closer to the destination, rather than determining the full path.

5.3.2.1. Large site, many hosts – minimum information

Berkeley is an example of a large site, i.e., more than two or three hosts. We have decided that the only reasonable philosophy in our environment is to designate one host as the guru for our site. It must be able to resolve any piece of mail it receives. The other sites should have the minimum amount of information they can get away with. In addition, any information they do have should be hints rather than solid information.

For example, a typical site on our local ether network is "monet." Monet has a list of known ethernet hosts; if it receives mail for any of them, it can do direct delivery. If it receives mail for any unknown host, it just passes it directly to "ucbvax," our master host. Ucbvax may determine that the host name is illegal and

reject the message, or may be able to do delivery. However, it is important to note that when a new ethernet host is added, the only host that *must* have its tables updated is ucbvax; the others *may* be updated as convenient, but this is not critical.

This picture is slightly muddied due to network connections that are not actually located on ucbvax. For example, our TCP connection is currently on "ucbarpa." However, monet *does not* know about this; the information is hidden totally between ucbvax and ucbarpa. Mail going from monet to a TCP host is transfered via the ethernet from monet to ucbvax, then via the ethernet from ucbvax to ucbarpa, and then is submitted to the Arpanet. Although this involves some extra hops, we feel this is an acceptable tradeoff.

An interesting point is that it would be possible to update monet to send TCP mail directly to ucbarpa if the load got too high; if monet failed to note a host as a TCP host it would go via ucbvax as before, and if monet incorrectly sent a message to ucbarpa it would still be sent by ucbarpa to ucbvax as before. The only problem that can occur is loops, as if ucbarpa thought that ucbvax had the TCP connection and vice versa. For this reason, updates should *always* happen to the master host first.

This philosophy results as much from the need to have a single source for the configuration files (typically built using *m4*(1) or some similar tool) as any logical need. Maintaining more than three separate tables by hand is essentially an impossible job.

5.3.2.2. Small site – complete information

A small site (two or three hosts) may find it more reasonable to have complete information at each host. This would require that each host know exactly where each network connection is, possibly including the names of each host on that network. As long as the site remains small and the the configuration remains relatively static, the update problem will probably not be too great.

5.3.2.3. Single host

This is in some sense the trivial case. The only major issue is trying to insure that you don't have to know too much about your environment. For example, if you have a UUCP connection you might find it useful to know about the names of hosts connected directly to you, but this is really not necessary since this may be determined from the syntax.

5.3.3. Relevant issues

The canonical form you use should almost certainly be as specified in the Arpanet protocols RFC819 and RFC822. Copies of these RFC's are included on the *sendmail* tape as *doc/rfc819.lpr* and *doc/rfc822.lpr*.

RFC822 describes the format of the mail message itself. *Sendmail* follows this RFC closely, to the extent that many of the standards described in this document can not be changed without changing the code. In particular, the following characters have special interpretations:

< > ( ) " \

Any attempt to use these characters for other than their RFC822 purpose in addresses is probably doomed to disaster.

RFC819 describes the specifics of the domain-based addressing. This is touched on in RFC822 as well. Essentially each host is given a name which is a right-to-left dot qualified pseudo-path from a distinguished root. The elements of the path need not be physical hosts; the domain is logical rather than physical. For example, at Berkeley one legal host is "a.cc.berkeley.arpa"; reading from right to left, "arpa" is a top level

domain (related to, but not limited to, the physical Arpanet), "berkeley" is both an Arpanet host and a logical domain which is actually interpreted by a host called ucbvax (which is actually just the "major" host for this domain), "cc" represents the Computer Center, (in this case a strictly logical entity), and "a" is a host in the Computer Center; this particular host happens to be connected via berknet, but other hosts might be connected via one of two ethernets or some other network.

Beware when reading RFC819 that there are a number of errors in it.

5.3.4. How to proceed

Once you have decided on a philosophy, it is worth examining the available configuration tables to decide if any of them are close enough to steal major parts of. Even under the worst of conditions, there is a fair amount of boiler plate that can be collected safely.

The next step is to build ruleset three. This will be the hardest part of the job. Beware of doing too much to the address in this ruleset, since anything you do will reflect through to the message. In particular, stripping of local domains is best deferred, since this can leave you with addresses with no domain spec at all. Since *sendmail* likes to append the sending domain to addresses with no domain, this can change the semantics of addresses. Also try to avoid fully qualifying domains in this ruleset. Although technically legal, this can lead to unpleasantly and unnecessarily long addresses reflected into messages. The Berkeley configuration files define ruleset nine to qualify domain names and strip local domains. This is called from ruleset zero to get all addresses into a cleaner form.

Once you have ruleset three finished, the other rulesets should be relatively trivial. If you need hints, examine the supplied configuration tables.

5.3.5. Testing the rewriting rules – the –bt flag

When you build a configuration table, you can do a certain amount of testing using the "test mode" of *sendmail*. For example, you could invoke *sendmail* as:

sendmail –bt –Ctest.cf

which would read the configuration file "test.cf" and enter test mode. In this mode, you enter lines of the form:

rwset address

where *rwset* is the rewriting set you want to use and *address* is an address to apply the set to. Test mode shows you the steps it takes as it proceeds, finally showing you the address it ends up with. You may use a comma separated list of rwsets for sequential application of rules to an input; ruleset three is always applied first. For example:

1,21,4 monet:bollard

first applies ruleset three to the input "monet:bollard." Ruleset one is then applied to the output of ruleset three, followed similarly by rulesets twenty-one and four.

If you need more detail, you can also use the "–d21" flag to turn on more debugging. For example,

sendmail –bt –d21.99

turns on an incredible amount of information; a single word address is probably going to print out several pages worth of information.

5.3.6. Building mailer descriptions

To add an outgoing mailer to your mail system, you will have to define the characteristics of the mailer.

Each mailer must have an internal name. This can be arbitrary, except that the names "local" and "prog" must be defined.

The pathname of the mailer must be given in the P field. If this mailer should be accessed via an IPC connection, use the string "[IPC]" instead.

The F field defines the mailer flags. You should specify an "f" or "r" flag to pass the name of the sender as a **–f** or **–r** flag respectively. These flags are only passed if they were passed to *sendmail*, so that mailers that give errors under some circumstances can be placated. If the mailer is not picky you can just specify "**–f $g**" in the argv template. If the mailer must be called as **root** the "S" flag should be given; this will not reset the userid before calling the mailer[3]. If this mailer is local (i.e., will perform final delivery rather than another network hop) the "l" flag should be given. Quote characters (backslashes and " marks) can be stripped from addresses if the "s" flag is specified; if this is not given they are passed through. If the mailer is capable of sending to more than one user on the same host in a single transaction the "m" flag should be stated. If this flag is on, then the argv template containing **$u** will be repeated for each unique user on a given host. The "e" flag will mark the mailer as being "expensive," which will cause *sendmail* to defer connection until a queue run[4].

An unusual case is the "C" flag. This flag applies to the mailer that the message is received from, rather than the mailer being sent to; if set, the domain spec of the sender (i.e., the "@host.domain" part) is saved and is appended to any addresses in the message that do not already contain a domain spec. For example, a message of the form:

From: eric@ucbarpa
To: wnj@monet, mckusick

will be modified to:

From: eric@ucbarpa
To: wnj@monet, mckusick@ucbarpa

if and only if the "C" flag is defined in the mailer corresponding to "eric@ucbarpa."

Other flags are described in Appendix C.

The S and R fields in the mailer description are per-mailer rewriting sets to be applied to sender and recipient addresses respectively. These are applied after the sending domain is appended and the general rewriting sets (numbers one and two) are applied, but before the output rewrite (ruleset four) is applied. A typical use is to append the current domain to addresses that do not already have a domain. For example, a header of the form:

From: eric

might be changed to be:

From: eric@ucbarpa

or

From: ucbvax!eric

depending on the domain it is being shipped into. These sets can also be used to do special purpose output rewriting in cooperation with ruleset four.

The E field defines the string to use as an end-of-line indication. A string containing only newline is the default. The usual backslash escapes (\r, \n, \f, \b) may be used.

---

[3] *Sendmail* must be running setuid to root for this to work.

[4] The "c" configuration option must be given for this to be effective.

Finally, an argv template is given as the E field. It may have embedded spaces. If there is no argv with a **$u** macro in it, *sendmail* will speak SMTP to the mailer. If the pathname for this mailer is "[IPC]," the argv should be

IPC $h [*port*]

where *port* is the optional port number to connect to.

For example, the specifications:

```
Mlocal, P=/bin/mail, F=rlsm  S=10, R=20, A=mail -d $u
Mether, P=[IPC],     F=meC, S=11, R=21, A=IPC $h, M=100000
```

specifies a mailer to do local delivery and a mailer for ethernet delivery. The first is called "local," is located in the file "/bin/mail," takes a picky **-r** flag, does local delivery, quotes should be stripped from addresses, and multiple users can be delivered at once; ruleset ten should be applied to sender addresses in the message and ruleset twenty should be applied to recipient addresses; the argv to send to a message will be the word "mail," the word "-d," and words containing the name of the receiving user. If a **-r** flag is inserted it will be between the words "mail" and "-d." The second mailer is called "ether," it should be connected to via an IPC connection, it can handle multiple users at once, connections should be deferred, and any domain from the sender address should be appended to any receiver name without a domain; sender addresses should be processed by ruleset eleven and recipient addresses by ruleset twenty-one. There is a 100,000 byte limit on messages passed through this mailer.

APPENDIX A

COMMAND LINE FLAGS

Arguments must be presented with flags before addresses. The flags are:

-f *addr* The sender's machine address is *addr*. This flag is ignored unless the real user is listed as a "trusted user" or if *addr* contains an exclamation point (because of certain restrictions in UUCP).

-r *addr* An obsolete form of **-f**.

-h *cnt* Sets the "hop count" to *cnt*. This represents the number of times this message has been processed by *sendmail* (to the extent that it is supported by the underlying networks). *Cnt* is incremented during processing, and if it reaches MAXHOP (currently 30) *sendmail* throws away the message with an error.

-F*name* Sets the full name of this user to *name*.

-n Don't do aliasing or forwarding.

-t Read the header for "To:", "Cc:", and "Bcc:" lines, and send to everyone listed in those lists. The "Bcc:" line will be deleted before sending. Any addresses in the argument vector will be deleted from the send list.

-b*x* Set operation mode to *x*. Operation modes are:

- m Deliver mail (default)
- a Run in arpanet mode (see below)
- s Speak SMTP on input side
- d Run as a daemon
- t Run in test mode
- v Just verify addresses, don't collect or deliver
- i Initialize the alias database
- p Print the mail queue
- z Freeze the configuration file

The special processing for the ARPANET includes reading the "From:" line from the header to find the sender, printing ARPANET style messages (preceded by three digit reply codes for compatibility with the FTP protocol [Neigus73, Postel74, Postel77]), and ending lines of error messages with <CRLF>.

-q*time* Try to process the queued up mail. If the time is given, a sendmail will run through the queue at the specified interval to deliver queued mail; otherwise, it only runs once.

-C*file* Use a different configuration file.

-d*level* Set debugging level.

-o*x value* Set option *x* to the specified *value*. These options are described in Appendix B.

There are a number of options that may be specified as primitive flags (provided for compatibility with *delivermail*). These are the e, i, m, and v options. Also, the f option may be specified as the **-s** flag.

APPENDIX B

CONFIGURATION OPTIONS

The following options may be set using the **–o** flag on the command line or the **O** line in the configuration file:

A*file* — Use the named *file* as the alias file. If no file is specified, use *aliases* in the current directory.

a — If set, wait for an "@:@" entry to exist in the alias database before starting up. If it does not appear in five minutes, rebuild the database.

c — If an outgoing mailer is marked as being expensive, don't connect immediately. This requires that queueing be compiled in, since it will depend on a queue run process to actually send the mail.

d*x* — Deliver in mode *x*. Legal modes are:

- i Deliver interactively (synchronously)
- b Deliver in background (asynchronously)
- q Just queue the message (deliver during queue run)

D — If set, rebuild the alias database if necessary and possible. If this option is not set, *sendmail* will never rebuild the alias database unless explicitly requested using **–bi**.

e*x* — Dispose of errors using mode *x*. The values for *x* are:

- p Print error messages (default)
- q No messages, just give exit status
- m Mail back errors
- w Write back errors (mail if user not logged in)
- e Mail back errors and give zero exit stat always

F*n* — The temporary file mode, in octal. 644 and 600 are good choices.

f — Save Unix-style "From" lines at the front of headers. Normally they are assumed redundant and discarded.

g*n* — Set the default group id for mailers to run in to *n*.

H*file* — Specify the help file for SMTP.

i — Ignore dots in incoming messages.

L*n* — Set the default log level to *n*.

M*x value* — Set the macro *x* to *value*. This is intended only for use from the command line.

m — Send to me too, even if I am in an alias expansion.

o — Assume that the headers may be in old format, i.e., spaces delimit names. This actually turns on an adaptive algorithm: if any recipient address contains a comma, parenthesis, or angle bracket, it will be assumed that commas already exist. If this flag is not on, only commas delimit names. Headers are always output with commas between the names.

Q*dir* — Use the named *dir* as the queue directory.

r*time* — Timeout reads after *time* interval.

S*file* — Log statistics in the named *file*.

s	Be super-safe when running things, i.e., always instantiate the queue file, even if you are going to attempt immediate delivery. *Sendmail* always instantiates the queue file before returning control the the client under any circumstances.
T*time*	Set the queue timeout to *time*. After this interval, messages that have not been successfully sent will be returned to the sender.
t*S,D*	Set the local timezone name to *S* for standard time and *D* for daylight time; this is only used under version six.
u*n*	Set the default userid for mailers to *n*. Mailers without the *S* flag in the mailer definition will run as this user.
v	Run in verbose mode.

APPENDIX C

MAILER FLAGS

The following flags may be set in the mailer description.

f The mailer wants a **–f** *from* flag, but only if this is a network forward operation (i.e., the mailer will give an error if the executing user does not have special permissions).

r Same as **f**, but sends a **–r** flag.

S Don't reset the userid before calling the mailer. This would be used in a secure environment where *sendmail* ran as root. This could be used to avoid forged addresses. This flag is suppressed if given from an "unsafe" environment (e.g, a user's mail.cf file).

n Do not insert a UNIX-style "From" line on the front of the message.

l This mailer is local (i.e., final delivery will be performed).

s Strip quote characters off of the address before calling the mailer.

m This mailer can send to multiple users on the same host in one transaction. When a **$u** macro occurs in the *argv* part of the mailer definition, that field will be repeated as necessary for all qualifying users.

F This mailer wants a "From:" header line.

D This mailer wants a "Date:" header line.

M This mailer wants a "Message-Id:" header line.

x This mailer wants a "Full-Name:" header line.

P This mailer wants a "Return-Path:" line.

u Upper case should be preserved in user names for this mailer.

h Upper case should be preserved in host names for this mailer.

A This is an Arpanet-compatible mailer, and all appropriate modes should be set.

U This mailer wants Unix-style "From" lines with the ugly UUCP-style "remote from <host>" on the end.

e This mailer is expensive to connect to, so try to avoid connecting normally; any necessary connection will occur during a queue run.

X This mailer want to use the hidden dot algorithm as specified in RFC821; basically, any line beginning with a dot will have an extra dot prepended (to be stripped at the other end). This insures that lines in the message containing a dot will not terminate the message prematurely.

L Limit the line lengths as specified in RFC821.

P Use the return-path in the SMTP "MAIL FROM:" command rather than just the return address; although this is required in RFC821, many hosts do not process return paths properly.

I This mailer will be speaking SMTP to another *sendmail* – as such it can use special protocol features. This option is not required (i.e., if this option is omitted the transmission will still operate successfully, although perhaps not as efficiently as possible).

C If mail is *received* from a mailer with this flag set, any addresses in the header that do not have an at sign ("@") after being rewritten by ruleset three will have the "@domain" clause from the sender tacked on. This allows mail with headers of the form:

From: usera@hosta
To: userb@hostb, userc

to be rewritten as:

From: usera@hosta
To: userb@hostb, userc@hosta

automatically.

APPENDIX D

OTHER CONFIGURATION

There are some configuration changes that can be made by recompiling *sendmail*. These are located in three places:

md/config.m4 These contain operating-system dependent descriptions. They are interpolated into the Makefiles in the *src* and *aux* directories. This includes information about what version of UNIX you are running, what libraries you have to include, etc.

src/conf.h Configuration parameters that may be tweaked by the installer are included in conf.h.

src/conf.c Some special routines and a few variables may be defined in conf.c. For the most part these are selected from the settings in conf.h.

Parameters in md/config.m4

The following compilation flags may be defined in the *m4CONFIG* macro in *md/config.m4* to define the environment in which you are operating.

V6 If set, this will compile a version 6 system, with 8-bit user id's, single character tty id's, etc.

VMUNIX If set, you will be assumed to have a Berkeley 4BSD or 4.1BSD, including the *vfork*(2) system call, special types defined in <sys/types.h> (e.g, u_char), etc.

If none of these flags are set, a version 7 system is assumed.

You will also have to specify what libraries to link with *sendmail* in the *m4LIBS* macro. Most notably, you will have to include if you are running a 4.1BSD system.

Parameters in src/conf.h

Parameters and compilation options are defined in conf.h. Most of these need not normally be tweaked; common parameters are all in sendmail.cf. However, the sizes of certain primitive vectors, etc., are included in this file. The numbers following the parameters are their default value.

MAXLINE [256] The maximum line length of any input line. If message lines exceed this length they will still be processed correctly; however, header lines, configuration file lines, alias lines, etc., must fit within this limit.

MAXNAME [128] The maximum length of any name, such as a host or a user name.

MAXFIELD [2500] The maximum total length of any header field, including continuation lines.

MAXPV [40] The maximum number of parameters to any mailer. This limits the number of recipients that may be passed in one transaction.

MAXHOP [30] When a message has been processed more than this number of times, sendmail rejects the message on the assumption that there has been an aliasing loop. This can be determined from the **-h** flag or by counting the number of trace fields (i.e, "Received:" lines) in the message header.

MAXATOM [100] The maximum number of atoms (tokens) in a single address. For example, the address "eric@Berkeley" is three atoms.

MAXMAILERS [25]
The maximum number of mailers that may be defined in the configuration file.

MAXRWSETS [30] The maximum number of rewriting sets that may be defined.

MAXPRIORITIES [25]
The maximum number of values for the "Precedence:" field that may be defined (using the **P** line in sendmail.cf).

MAXTRUST [30] The maximum number of trusted users that may be defined (using the **T** line in sendmail.cf).

A number of other compilation options exist. These specify whether or not specific code should be compiled in.

DBM If set, the "DBM" package in UNIX is used (see DBM(3X) in [UNIX80]). If not set, a much less efficient algorithm for processing aliases is used.

DEBUG If set, debugging information is compiled in. To actually get the debugging output, the **–d** flag must be used.

LOG If set, the *syslog* routine in use at some sites is used. This makes an informational log record for each message processed, and makes a higher priority log record for internal system errors.

QUEUE This flag should be set to compile in the queueing code. If this is not set, mailers must accept the mail immediately or it will be returned to the sender.

SMTP If set, the code to handle user and server SMTP will be compiled in. This is only necessary if your machine has some mailer that speaks SMTP.

DAEMON If set, code to run a daemon is compiled in. This code is for 4.2BSD if the NVMUNIX flag is specified; otherwise, 4.1a BSD code is used. Beware however that there are bugs in the 4.1a code that make it impossible for **sendmail** to work correctly under heavy load.

UGLYUUCP If you have a UUCP host adjacent to you which is not running a reasonable version of *rmail*, you will have to set this flag to include the "remote from sysname" info on the from line. Otherwise, UUCP gets confused about where the mail came from.

NOTUNIX If you are using a non-UNIX mail format, you can set this flag to turn off special processing of UNIX-style "From " lines.

Configuration in src/conf.c

Not all header semantics are defined in the configuration file. Header lines that should only be included by certain mailers (as well as other more obscure semantics) must be specified in the *HdrInfo* table in *conf.c*. This table contains the header name (which should be in all lower case) and a set of header control flags (described below), The flags are:

H_ACHECK Normally when the check is made to see if a header line is compatible with a mailer, *sendmail* will not delete an existing line. If this flag is set, *sendmail* will delete even existing header lines. That is, if this bit is set and the mailer does not have flag bits set that intersect with the required mailer flags in the header definition in sendmail.cf, the header line is *always* deleted.

H_EOH If this header field is set, treat it like a blank line, i.e., it will signal the end of the header and the beginning of the message text.

H_FORCE Add this header entry even if one existed in the message before. If a header entry does not have this bit set, *sendmail* will not add another header line if a header line of this name already existed. This would normally be used to stamp the message by everyone who handled it.

H_TRACE If set, this is a timestamp (trace) field. If the number of trace fields in a message exceeds a preset amount the message is returned on the assumption that it has an aliasing loop.

H_RCPT If set, this field contains recipient addresses. This is used by the **-t** flag to determine who to send to when it is collecting recipients from the message.

H_FROM This flag indicates that this field specifies a sender. The order of these fields in the *HdrInfo* table specifies *sendmail's* preference for which field to return error messages to.

Let's look at a sample *HdrInfo* specification:

```
struct hdrinfo          HdrInfo[] =
{
        /* originator fields, most to least significant */
    "resent-sender",    H_FROM,
    "resent-from",      H_FROM,
    "sender",           H_FROM,
    "from",             H_FROM,
    "full-name",        H_ACHECK,
        /* destination fields */
    "to",               H_RCPT,
    "resent-to",        H_RCPT,
    "cc",               H_RCPT,
        /* message identification and control */
    "message",          H_EOH,
    "text",             H_EOH,
        /* trace fields */
    "received",         H_TRACE|H_FORCE,

    NULL,               0,
};
```

This structure indicates that the "To:", "Resent-To:", and "Cc:" fields all specify recipient addresses. Any "Full-Name:" field will be deleted unless the required mailer flag (indicated in the configuration file) is specified. The "Message:" and "Text:" fields will terminate the header; these are specified in new protocols [NBS80] or used by random dissenters around the network world. The "Received:" field will always be added, and can be used to trace messages.

There are a number of important points here. First, header fields are not added automatically just because they are in the *HdrInfo* structure; they must be specified in the configuration file in order to be added to the message. Any header fields mentioned in the configuration file but not mentioned in the *HdrInfo* structure have default processing performed; that is, they are added unless they were in the message already. Second, the *HdrInfo* structure only specifies cliched processing; certain headers are processed specially by ad hoc code regardless of the status specified in *HdrInfo*. For example, the "Sender:" and "From:" fields are always scanned on ARPANET mail to determine the sender; this is used to perform the "return to sender" function. The "From:" and "Full-Name:" fields are used to determine the full name of the sender if possible; this is stored in the macro **$x** and used in a number of ways.

The file *conf.c* also contains the specification of ARPANET reply codes. There are four classifications these fall into:

```
char  Arpa_Info[] =     "050";  /* arbitrary info */
char  Arpa_TSyserr[] =  "455";  /* some (transient) system error */
char  Arpa_PSyserr[] =  "554";  /* some (transient) system error */
char  Arpa_Usrerr[] =   "554";  /* some (fatal) user error */
```

The class *Arpa_Info* is for any information that is not required by the protocol, such as forwarding information. *Arpa_TSyserr* and *Arpa_PSyserr* is printed by the *syserr* routine. TSyserr is printed out for transient errors, whereas PSyserr is printed for permanent errors; the distinction is made based on the value of *errno*. Finally, *Arpa_Usrerr* is the result of a user error and is generated by the *usrerr* routine; these are generated when the user has specified something wrong, and hence the error is permanent, i.e., it will not work simply by resubmitting the request.

If it is necessary to restrict mail through a relay, the *checkcompat* routine can be modified. This routine is called for every recipient address. It can return **TRUE** to indicate that the address is acceptable and mail processing will continue, or it can return **FALSE** to reject the recipient. If it returns false, it is up to *checkcompat* to print an error message (using *usrerr*) saying why the message is rejected. For example, *checkcompat* could read:

```
bool
checkcompat(to)
    register ADDRESS *to;
{
    if (MsgSize > 50000 && to->q_mailer != LocalMailer)
    {
        usrerr("Message too large for non-local delivery");
        NoReturn = TRUE;
        return (FALSE);
    }
    return (TRUE);
}
```

This would reject messages greater than 50000 bytes unless they were local. The *NoReturn* flag can be sent to supress the return of the actual body of the message in the error return. The actual use of this routine is highly dependent on the implementation, and use should be limited.

APPENDIX E

SUMMARY OF SUPPORT FILES

This is a summary of the support files that *sendmail* creates or generates.

/usr/lib/sendmail
: The binary of *sendmail*.

/usr/bin/newaliases
: A link to /usr/lib/sendmail; causes the alias database to be rebuilt. Running this program is completely equivalent to giving *sendmail* the **-bi** flag.

/usr/bin/mailq
: Prints a listing of the mail queue. This program is equivalent to using the **-bp** flag to *sendmail*.

/usr/lib/sendmail.cf
: The configuration file, in textual form.

/usr/lib/sendmail.fc
: The configuration file represented as a memory image.

/usr/lib/sendmail.hf
: The SMTP help file.

/usr/lib/sendmail.st
: A statistics file; need not be present.

/usr/lib/aliases
: The textual version of the alias file.

/usr/lib/aliases.{pag,dir}
: The alias file in *dbm*(3) format.

/etc/syslog
: The program to do logging.

/etc/syslog.conf
: The configuration file for syslog.

/etc/syslog.pid
: Contains the process id of the currently running syslog.

/usr/spool/mqueue
: The directory in which the mail queue and temporary files reside.

/usr/spool/mqueue/qf*
: Control (queue) files for messages.

/usr/spool/mqueue/df*
: Data files.

/usr/spool/mqueue/lf*
: Lock files

/usr/spool/mqueue/tf*
: Temporary versions of the qf files, used during queue file rebuild.

/usr/spool/mqueue/nf*
: A file used when creating a unique id.

/usr/spool/mqueue/xf*
: A transcript of the current session.

SENDMAIL – An Internetwork Mail Router

Eric Allman†

Britton-Lee, Inc.
1919 Addison Street, Suite 105.
Berkeley, California 94704.

ABSTRACT

Routing mail through a heterogenous internet presents many new problems. Among the worst of these is that of address mapping. Historically, this has been handled on an *ad hoc* basis. However, this approach has become unmanageable as internets grow.

Sendmail acts a unified "post office" to which all mail can be submitted. Address interpretation is controlled by a production system, which can parse both domain-based addressing and old-style *ad hoc* addresses. The production system is powerful enough to rewrite addresses in the message header to conform to the standards of a number of common target networks, including old (NCP/RFC733) Arpanet, new (TCP/RFC822) Arpanet, UUCP, and Phonenet. Sendmail also implements an SMTP server, message queueing, and aliasing.

Sendmail implements a general internetwork mail routing facility, featuring aliasing and forwarding, automatic routing to network gateways, and flexible configuration.

In a simple network, each node has an address, and resources can be identified with a host-resource pair; in particular, the mail system can refer to users using a host-username pair. Host names and numbers have to be administered by a central authority, but usernames can be assigned locally to each host.

In an internet, multiple networks with different characterstics and managements must communicate. In particular, the syntax and semantics of resource identification change. Certain special cases can be handled trivially by *ad hoc* techniques, such as providing network names that appear local to hosts on other networks, as with the Ethernet at Xerox PARC. However, the general case is extremely complex. For example, some networks require point-to-point routing, which simplifies the database update problem since only adjacent hosts must be entered into the system tables, while others use end-to-end addressing. Some networks use a left-associative syntax and others use a right-associative syntax, causing ambiguity in mixed addresses.

Internet standards seek to eliminate these problems. Initially, these proposed expanding the address pairs to address triples, consisting of {network, host, resource} triples. Network numbers must be universally agreed upon, and hosts can be assigned locally on each network. The user-level presentation was quickly expanded to address domains, comprised of a local resource identification and a hierarchical domain specification with a common static root. The domain technique separates the issue of physical versus logical addressing. For example, an address of the form "eric@a.cc.berkeley.arpa" describes only the logical organization of the address space.

Sendmail is intended to help bridge the gap between the totally *ad hoc* world of networks that know nothing of each other and the clean, tightly-coupled world of unique network numbers. It can accept old arbitrary address syntaxes, resolving ambiguities using heuristics specified by the

†A considerable part of this work was done while under the employ of the INGRES Project at the University of California at Berkeley.

system administrator, as well as domain-based addressing. It helps guide the conversion of message formats between disparate networks. In short, *sendmail* is designed to assist a graceful transition to consistent internetwork addressing schemes.

Section 1 discusses the design goals for *sendmail*. Section 2 gives an overview of the basic functions of the system. In section 3, details of usage are discussed. Section 4 compares *sendmail* to other internet mail routers, and an evaluation of *sendmail* is given in section 5, including future plans.

1. DESIGN GOALS

Design goals for *sendmail* include:

(1) Compatibility with the existing mail programs, including Bell version 6 mail, Bell version 7 mail [UNIX83], Berkeley *Mail* [Shoens79], BerkNet mail [Schmidt79], and hopefully UUCP mail [Nowitz78a, Nowitz78b]. ARPANET mail [Crocker77a, Postel77] was also required.

(2) Reliability, in the sense of guaranteeing that every message is correctly delivered or at least brought to the attention of a human for correct disposal; no message should ever be completely lost. This goal was considered essential because of the emphasis on mail in our environment. It has turned out to be one of the hardest goals to satisfy, especially in the face of the many anomalous message formats produced by various ARPANET sites. For example, certain sites generate improperly formated addresses, occasionally causing error-message loops. Some hosts use blanks in names, causing problems with UNIX mail programs that assume that an address is one word. The semantics of some fields are interpreted slightly differently by different sites. In summary, the obscure features of the ARPANET mail protocol really *are* used and are difficult to support, but must be supported.

(3) Existing software to do actual delivery should be used whenever possible. This goal derives as much from political and practical considerations as technical.

(4) Easy expansion to fairly complex environments, including multiple connections to a single network type (such as with multiple UUCP or Ether nets [Metcalfe76]). This goal requires consideration of the contents of an address as well as its syntax in order to determine which gateway to use. For example, the ARPANET is bringing up the TCP protocol to replace the old NCP protocol. No host at Berkeley runs both TCP and NCP, so it is necessary to look at the ARPANET host name to determine whether to route mail to an NCP gateway or a TCP gateway.

(5) Configuration should not be compiled into the code. A single compiled program should be able to run as is at any site (barring such basic changes as the CPU type or the operating system). We have found this seemingly unimportant goal to be critical in real life. Besides the simple problems that occur when any program gets recompiled in a different environment, many sites like to "fiddle" with anything that they will be recompiling anyway.

(6) *Sendmail* must be able to let various groups maintain their own mailing lists, and let individuals specify their own forwarding, without modifying the system alias file.

(7) Each user should be able to specify which mailer to execute to process mail being delivered for him. This feature allows users who are using specialized mailers that use a different format to build their environment without changing the system, and facilitates specialized functions (such as returning an "I am on vacation" message).

(8) Network traffic should be minimized by batching addresses to a single host where possible, without assistance from the user.

These goals motivated the architecture illustrated in figure 1. The user interacts with a mail generating and sending program. When the mail is created, the generator calls *sendmail*,

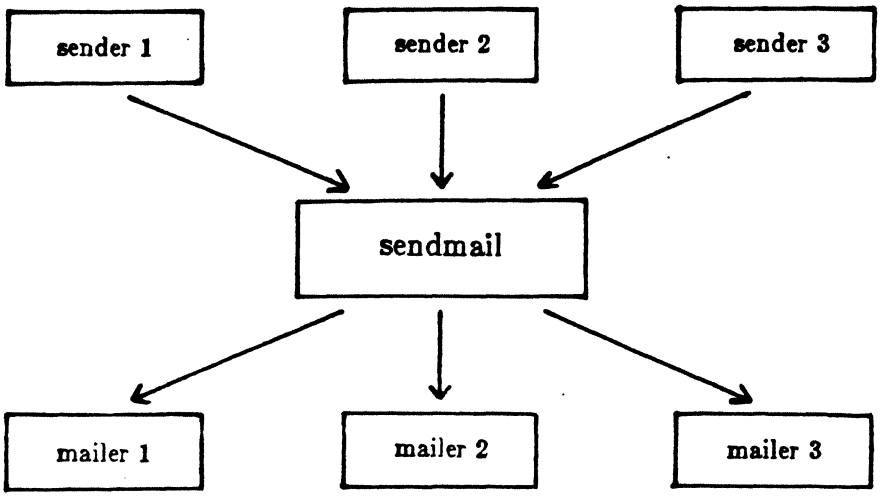


Figure 1 – Sendmail System Structure.

which routes the message to the correct mailer(s). Since some of the senders may be network servers and some of the mailers may be network clients, *sendmail* may be used as an internet mail gateway.

2. OVERVIEW

2.1. System Organization

Sendmail neither interfaces with the user nor does actual mail delivery. Rather, it collects a message generated by a user interface program (UIP) such as Berkeley *Mail*, MS [Crocker77b], or MH [Borden79], edits the message as required by the destination network, and calls appropriate mailers to do mail delivery or queueing for network transmission[1]. This discipline allows the insertion of new mailers at minimum cost. In this sense *sendmail* resembles the Message Processing Module (MPM) of [Postel79b].

2.2. Interfaces to the Outside World

There are three ways *sendmail* can communicate with the outside world, both in receiving and in sending mail. These are using the conventional UNIX argument vector/return status, speaking SMTP over a pair of UNIX pipes, and speaking SMTP over an interprocess(or) channel.

2.2.1. Argument vector/exit status

This technique is the standard UNIX method for communicating with the process. A list of recipients is sent in the argument vector, and the message body is sent on the standard input. Anything that the mailer prints is simply collected and sent back to the sender if there were any problems. The exit status from the mailer is collected after the message is sent, and a diagnostic is printed if appropriate.

[1]except when mailing to a file, when *sendmail* does the delivery directly.

2.2.2. SMTP over pipes

The SMTP protocol [Postel82] can be used to run an interactive lock-step interface with the mailer. A subprocess is still created, but no recipient addresses are passed to the mailer via the argument list. Instead, they are passed one at a time in commands sent to the processes standard input. Anything appearing on the standard output must be a reply code in a special format.

2.2.3. SMTP over an IPC connection

This technique is similar to the previous technique, except that it uses a 4.2BSD IPC channel [UNIX83]. This method is exceptionally flexible in that the mailer need not reside on the same machine. It is normally used to connect to a sendmail process on another machine.

2.3. Operational Description

When a sender wants to send a message, it issues a request to *sendmail* using one of the three methods described above. *Sendmail* operates in two distinct phases. In the first phase, it collects and stores the message. In the second phase, message delivery occurs. If there were errors during processing during the second phase, *sendmail* creates and returns a new message describing the error and/or returns an status code telling what went wrong.

2.3.1. Argument processing and address parsing

If *sendmail* is called using one of the two subprocess techniques, the arguments are first scanned and option specifications are processed. Recipient addresses are then collected, either from the command line or from the SMTP RCPT command, and a list of recipients is created. Aliases are expanded at this step, including mailing lists. As much validation as possible of the addresses is done at this step: syntax is checked, and local addresses are verified, but detailed checking of host names and addresses is deferred until delivery. Forwarding is also performed as the local addresses are verified.

Sendmail appends each address to the recipient list after parsing. When a name is aliased or forwarded, the old name is retained in the list, and a flag is set that tells the delivery phase to ignore this recipient. This list is kept free from duplicates, preventing alias loops and duplicate messages deliverd to the same recipient, as might occur if a person is in two groups.

2.3.2. Message collection

Sendmail then collects the message. The message should have a header at the beginning. No formatting requirements are imposed on the message except that they must be lines of text (i.e., binary data is not allowed). The header is parsed and stored in memory, and the body of the message is saved in a temporary file.

To simplify the program interface, the message is collected even if no addresses were valid. The message will be returned with an error.

2.3.3. Message delivery

For each unique mailer and host in the recipient list, *sendmail* calls the appropriate mailer. Each mailer invocation sends to all users receiving the message on one host. Mailers that only accept one recipient at a time are handled properly.

The message is sent to the mailer using one of the same three interfaces used to submit a message to sendmail. Each copy of the message is prepended by a customized header. The mailer status code is caught and checked, and a suitable error message given as appropriate. The exit code must conform to a system standard or a generic message ("Service unavailable") is given.

2.3.4. Queueing for retransmission

If the mailer returned an status that indicated that it might be able to handle the mail later, *sendmail* will queue the mail and try again later.

2.3.5. Return to sender

If errors occur during processing, *sendmail* returns the message to the sender for retransmission. The letter can be mailed back or written in the file "dead.letter" in the sender's home directory[2].

2.4. Message Header Editing

Certain editing of the message header occurs automatically. Header lines can be inserted under control of the configuration file. Some lines can be merged; for example, a "From:" line and a "Full-name:" line can be merged under certain circumstances.

2.5. Configuration File

Almost all configuration information is read at runtime from an ASCII file, encoding macro definitions (defining the value of macros used internally), header declarations (telling sendmail the format of header lines that it will process specially, i.e., lines that it will add or reformat), mailer definitions (giving information such as the location and characteristics of each mailer), and address rewriting rules (a limited production system to rewrite addresses which is used to parse and rewrite the addresses).

To improve performance when reading the configuration file, a memory image can be provided. This provides a "compiled" form of the configuration file.

3. USAGE AND IMPLEMENTATION

3.1. Arguments

Arguments may be flags and addresses. Flags set various processing options. Following flag arguments, address arguments may be given, unless we are running in SMTP mode. Addresses follow the syntax in RFC822 [Crocker82] for ARPANET address formats. In brief, the format is:

(1) Anything in parentheses is thrown away (as a comment).

(2) Anything in angle brackets ("< >") is preferred over anything else. This rule implements the ARPANET standard that addresses of the form

user name <machine-address>

will send to the electronic "machine-address" rather than the human "user name."

(3) Double quotes (") quote phrases; backslashes quote characters. Backslashes are more powerful in that they will cause otherwise equivalent phrases to compare differently – for example, *user* and "*user*" are equivalent, but *user* is different from either of them.

Parentheses, angle brackets, and double quotes must be properly balanced and nested. The rewriting rules control remaining parsing[3].

3.2. Mail to Files and Programs

Files and programs are legitimate message recipients. Files provide archival storage of messages, useful for project administration and history. Programs are useful as recipients in

[2]Obviously, if the site giving the error is not the originating site, the only reasonable option is to mail back to the sender. Also, there are many more error disposition options, but they only effect the error message – the "return to sender" function is always handled in one of these two ways.

[3]Disclaimer: Some special processing is done after rewriting local names; see below.

a variety of situations, for example, to maintain a public repository of systems messages (such as the Berkeley *msgs* program, or the MARS system [Sattley78]).

Any address passing through the initial parsing algorithm as a local address (i.e, not appearing to be a valid address for another mailer) is scanned for two special cases. If prefixed by a vertical bar ("|") the rest of the address is processed as a shell command. If the user name begins with a slash mark ("/") the name is used as a file name, instead of a login name.

Files that have setuid or setgid bits set but no execute bits set have those bits honored if *sendmail* is running as root.

3.3. Aliasing, Forwarding, Inclusion

Sendmail reroutes mail three ways. Aliasing applies system wide. Forwarding allows each user to reroute incoming mail destined for that account. Inclusion directs *sendmail* to read a file for a list of addresses, and is normally used in conjunction with aliasing.

3.3.1. Aliasing

Aliasing maps names to address lists using a system-wide file. This file is indexed to speed access. Only names that parse as local are allowed as aliases; this guarantees a unique key (since there are no nicknames for the local host).

3.3.2. Forwarding

After aliasing, recipients that are local and valid are checked for the existence of a ".forward" file in their home directory. If it exists, the message is *not* sent to that user, but rather to the list of users in that file. Often this list will contain only one address, and the feature will be used for network mail forwarding.

Forwarding also permits a user to specify a private incoming mailer. For example, forwarding to:

"|/usr/local/newmail myname"

will use a different incoming mailer.

3.3.3. Inclusion

Inclusion is specified in RFC 733 [Crocker77a] syntax:

:Include: pathname

An address of this form reads the file specified by *pathname* and sends to all users listed in that file.

The intent is *not* to support direct use of this feature, but rather to use this as a subset of aliasing. For example, an alias of the form:

project: :include:/usr/project/userlist

is a method of letting a project maintain a mailing list without interaction with the system administration, even if the alias file is protected.

It is not necessary to rebuild the index on the alias database when a :include: list is changed.

3.4. Message Collection

Once all recipient addresses are parsed and verified, the message is collected. The message comes in two parts: a message header and a message body, separated by a blank line.

The header is formatted as a series of lines of the form

field-name: field-value

Field-value can be split across lines by starting the following lines with a space or a tab. Some header fields have special internal meaning, and have appropriate special processing. Other headers are simply passed through. Some header fields may be added automatically, such as time stamps.

The body is a series of text lines. It is completely uninterpreted and untouched, except that lines beginning with a dot have the dot doubled when transmitted over an SMTP channel. This extra dot is stripped by the receiver.

3.5. Message Delivery

The send queue is ordered by receiving host before transmission to implement message batching. Each address is marked as it is sent so rescanning the list is safe. An argument list is built as the scan proceeds. Mail to files is detected during the scan of the send list. The interface to the mailer is performed using one of the techniques described in section 2.2.

After a connection is established, *sendmail* makes the per-mailer changes to the header and sends the result to the mailer. If any mail is rejected by the mailer, a flag is set to invoke the return-to-sender function after all delivery completes.

3.6. Queued Messages

If the mailer returns a "temporary failure" exit status, the message is queued. A control file is used to describe the recipients to be sent to and various other parameters. This control file is formatted as a series of lines, each describing a sender, a recipient, the time of submission, or some other salient parameter of the message. The header of the message is stored in the control file, so that the associated data file in the queue is just the temporary file that was originally collected.

3.7. Configuration

Configuration is controlled primarily by a configuration file read at startup. *Sendmail* should not need to be recomplied except

(1) To change operating systems (V6, V7/32V, 4BSD).

(2) To remove or insert the DBM (UNIX database) library.

(3) To change ARPANET reply codes.

(4) To add headers fields requiring special processing.

Adding mailers or changing parsing (i.e., rewriting) or routing information does not require recompilation.

If the mail is being sent by a local user, and the file ".mailcf" exists in the sender's home directory, that file is read as a configuration file after the system configuration file. The primary use of this feature is to add header lines.

The configuration file encodes macro definitions, header definitions, mailer definitions, rewriting rules, and options.

3.7.1. Macros

Macros can be used in three ways. Certain macros transmit unstructured textual information into the mail system, such as the name *sendmail* will use to identify itself in error messages. Other macros transmit information from *sendmail* to the configuration file for use in creating other fields (such as argument vectors to mailers); e.g., the name of the sender, and the host and user of the recipient. Other macros are unused internally, and can be used as shorthand in the configuration file.

3.7.2. Header declarations

Header declarations inform *sendmail* of the format of known header lines. Knowledge of a few header lines is built into *sendmail*, such as the "From:" and "Date:" lines.

Most configured headers will be automatically inserted in the outgoing message if they don't exist in the incoming message. Certain headers are suppressed by some mailers.

3.7.3. Mailer declarations

Mailer declarations tell *sendmail* of the various mailers available to it. The definition specifies the internal name of the mailer, the pathname of the program to call, some flags associated with the mailer, and an argument vector to be used on the call; this vector is macro-expanded before use.

3.7.4. Address rewriting rules

The heart of address parsing in *sendmail* is a set of rewriting rules. These are an ordered list of pattern-replacement rules, (somewhat like a production system, except that order is critical), which are applied to each address. The address is rewritten textually until it is either rewritten into a special canonical form (i.e., a (mailer, host, user) 3-tuple, such as {arpanet, usc-isif, postel} representing the address "postel@usc-isif"), or it falls off the end. When a pattern matches, the rule is reapplied until it fails.

The configuration file also supports the editing of addresses into different formats. For example, an address of the form:

ucsfcgl!tef

might be mapped into:

tef@ucsfcgl.UUCP

to conform to the domain syntax. Translations can also be done in the other direction.

3.7.5. Option setting

There are several options that can be set from the configuration file. These include the pathnames of various support files, timeouts, default modes, etc.

4. COMPARISON WITH OTHER MAILERS

4.1. Delivermail

Sendmail is an outgrowth of *delivermail*. The primary differences are:

(1) Configuration information is not compiled in. This change simplifies many of the problems of moving to other machines. It also allows easy debugging of new mailers.

(2) Address parsing is more flexible. For example, *delivermail* only supported one gateway to any network, whereas *sendmail* can be sensitive to host names and reroute to different gateways.

(3) Forwarding and :include: features eliminate the requirement that the system alias file be writable by any user (or that an update program be written, or that the system administration make all changes).

(4) *Sendmail* supports message batching across networks when a message is being sent to multiple recipients.

(5) A mail queue is provided in *sendmail*. Mail that cannot be delivered immediately but can potentially be delivered later is stored in this queue for a later retry. The queue also provides a buffer against system crashes; after the message has been collected it

may be reliably redelivered even if the system crashes during the initial delivery.

(6) *Sendmail* uses the networking support provided by 4.2BSD to provide a direct interface networks such as the ARPANET and/or Ethernet using SMTP (the Simple Mail Transfer Protocol) over a TCP/IP connection.

4.2. MMDF

MMDF [Crocker79] spans a wider problem set than *sendmail*. For example, the domain of MMDF includes a "phone network" mailer, whereas *sendmail* calls on preexisting mailers in most cases.

MMDF and *sendmail* both support aliasing, customized mailers, message batching, automatic forwarding to gateways, queueing, and retransmission. MMDF supports two-stage timeout, which *sendmail* does not support.

The configuration for MMDF is compiled into the code[4].

Since MMDF does not consider backwards compatibility as a design goal, the address parsing is simpler but much less flexible.

It is somewhat harder to integrate a new channel[5] into MMDF. In particular, MMDF must know the location and format of host tables for all channels, and the channel must speak a special protocol. This allows MMDF to do additional verification (such as verifying host names) at submission time.

MMDF strictly separates the submission and delivery phases. Although *sendmail* has the concept of each of these stages, they are integrated into one program, whereas in MMDF they are split into two programs.

4.3. Message Processing Module

The Message Processing Module (MPM) discussed by Postel [Postel79b] matches *sendmail* closely in terms of its basic architecture. However, like MMDF, the MPM includes the network interface software as part of its domain.

MPM also postulates a duplex channel to the receiver, as does MMDF, thus allowing simpler handling of errors by the mailer than is possible in *sendmail*. When a message queued by *sendmail* is sent, any errors must be returned to the sender by the mailer itself. Both MPM and MMDF mailers can return an immediate error response, and a single error processor can create an appropriate response.

MPM prefers passing the message as a structured object, with type-length-value tuples[6]. Such a convention requires a much higher degree of cooperation between mailers than is required by *sendmail*. MPM also assumes a universally agreed upon internet name space (with each address in the form of a net-host-user tuple), which *sendmail* does not.

5. EVALUATIONS AND FUTURE PLANS

Sendmail is designed to work in a nonhomogeneous environment. Every attempt is made to avoid imposing unnecessary constraints on the underlying mailers. This goal has driven much of the design. One of the major problems has been the lack of a uniform address space, as postulated in [Postel79a] and [Postel79b].

A nonuniform address space implies that a path will be specified in all addresses, either explicitly (as part of the address) or implicitly (as with implied forwarding to gateways). This restriction has the unpleasant effect of making replying to messages exceedingly difficult, since

[4]Dynamic configuration tables are currently being considered for MMDF; allowing the installer to select either compiled or dynamic tables.

[5]The MMDF equivalent of a *sendmail* "mailer."

[6]This is similar to the NBS standard.

there is no one "address" for any person, but only a way to get there from wherever you are.

Interfacing to mail programs that were not initially intended to be applied in an internet environment has been amazingly successful, and has reduced the job to a manageable task.

Sendmail has knowledge of a few difficult environments built in. It generates ARPANET FTP/SMTP compatible error messages (prepended with three-digit numbers [Neigus73, Postel74, Postel82]) as necessary, optionally generates UNIX-style "From" lines on the front of messages for some mailers, and knows how to parse the same lines on input. Also, error handling has an option customized for BerkNet.

The decision to avoid doing any type of delivery where possible (even, or perhaps especially, local delivery) has turned out to be a good idea. Even with local delivery, there are issues of the location of the mailbox, the format of the mailbox, the locking protocol used, etc., that are best decided by other programs. One surprisingly major annoyance in many internet mailers is that the location and format of local mail is built in. The feeling seems to be that local mail is so common that it should be efficient. This feeling is not born out by our experience; on the contrary, the location and format of mailboxes seems to vary widely from system to system.

The ability to automatically generate a response to incoming mail (by forwarding mail to a program) seems useful ("I am on vacation until late August....") but can create problems such as forwarding loops (two people on vacation whose programs send notes back and forth, for instance) if these programs are not well written. A program could be written to do standard tasks correctly, but this would solve the general case.

It might be desirable to implement some form of load limiting. I am unaware of any mail system that addresses this problem, nor am I aware of any reasonable solution at this time.

The configuration file is currently practically inscrutable; considerable convenience could be realized with a higher-level format.

It seems clear that common protocols will be changing soon to accommodate changing requirements and environments. These changes will include modifications to the message header (e.g., [NBS80]) or to the body of the message itself (such as for multimedia messages [Postel80]). Experience indicates that these changes should be relatively trivial to integrate into the existing system.

In tightly coupled environments, it would be nice to have a name server such as Grapvine [Birrell82] integrated into the mail system. This would allow a site such as "Berkeley" to appear as a single host, rather than as a collection of hosts, and would allow people to move transparently among machines without having to change their addresses. Such a facility would require an automatically updated database and some method of resolving conflicts. Ideally this would be effective even without all hosts being under a single management. However, it is not clear whether this feature should be integrated into the aliasing facility or should be considered a "value added" feature outside *sendmail* itself.

As a more interesting case, the CSNET name server [Solomon81] provides an facility that goes beyond a single tightly-coupled environment. Such a facility would normally exist outside of *sendmail* however.

ACKNOWLEDGEMENTS

Thanks are due to Kurt Shoens for his continual cheerful assistance and good advice, Bill Joy for pointing me in the correct direction (over and over), and Mark Horton for more advice, prodding, and many of the good ideas. Kurt and Eric Schmidt are to be credited for using *delivermail* as a server for their programs (*Mail* and BerkNet respectively) before any sane person should have, and making the necessary modifications promptly and happily. Eric gave me considerable advice about the perils of network software which saved me an unknown amount of work and grief. Mark did the original implementation of the DBM version of aliasing, installed the VFORK code, wrote the current version of *rmail*, and was the person who really convinced me to put the work

into *delivermail* to turn it into *sendmail*. Kurt deserves accolades for using *sendmail* when I was myself afraid to take the risk; how a person can continue to be so enthusiastic in the face of so much bitter reality is beyond me.

Kurt, Mark, Kirk McKusick, Marvin Solomon, and many others have reviewed this paper, giving considerable useful advice.

Special thanks are reserved for Mike Stonebraker at Berkeley and Bob Epstein at Britton-Lee, who both knowingly allowed me to put so much work into this project when there were so many other things I really should have been working on.

REFERENCES

[Birrell82] Birrell, A. D., Levin, R., Needham, R. M., and Schroeder, M. D., "Grapevine: An Exercise in Distributed Computing." In *Comm. A.C.M. 25,* 4, April 82.

[Borden79] Borden, S., Gaines, R. S., and Shapiro, N. Z., *The MH Message Handling System: Users' Manual.* R-2367-PAF. Rand Corporation. October 1979.

[Crocker77a] Crocker, D. H., Vittal, J. J., Pogran, K. T., and Henderson, D. A. Jr., *Standard for the Format of ARPA Network Text Messages.* RFC 733, NIC 41952. In [Feinler78]. November 1977.

[Crocker77b] Crocker, D. H., *Framework and Functions of the MS Personal Message System.* R-2134-ARPA, Rand Corporation, Santa Monica, California. 1977.

[Crocker79] Crocker, D. H., Szurkowski, E. S., and Farber, D. J., *An Internetwork Memo Distribution Facility - MMDF.* 6th Data Communication Symposium, Asilomar. November 1979.

[Crocker82] Crocker, D. H., *Standard for the Format of Arpa Internet Text Messages.* RFC 822. Network Information Center, SRI International, Menlo Park, California. August 1982.

[Metcalfe76] Metcalfe, R., and Boggs, D., "Ethernet: Distributed Packet Switching for Local Computer Networks", *Communications of the ACM 19,* 7. July 1976.

[Feinler78] Feinler, E., and Postel, J. (eds.), *ARPANET Protocol Handbook.* NIC 7104, Network Information Center, SRI International, Menlo Park, California. 1978.

[NBS80] National Bureau of Standards, *Specification of a Draft Message Format Standard.* Report No. ICST/CBOS 80-2. October 1980.

[Neigus73] Neigus, N., *File Transfer Protocol for the ARPA Network.* RFC 542, NIC 17759. In [Feinler78]. August, 1973.

[Nowitz78a] Nowitz, D. A., and Lesk, M. E., *A Dial-Up Network of UNIX Systems.* Bell Laboratories. In UNIX Programmer's Manual, Seventh Edition, Volume 2. August, 1978.

[Nowitz78b] Nowitz, D. A., *Uucp Implementation Description.* Bell Laboratories. In UNIX Programmer's Manual, Seventh Edition, Volume 2. October, 1978.

[Postel74] Postel, J., and Neigus, N., Revised FTP Reply Codes. RFC 640, NIC 30843. In [Feinler78]. June, 1974.

[Postel77] Postel, J., *Mail Protocol.* NIC 29588. In [Feinler78]. November 1977.

[Postel79a] Postel, J., *Internet Message Protocol.* RFC 753, IEN 85. Network Information Center, SRI International, Menlo Park, California. March 1979.

[Postel79b] Postel, J. B., *An Internetwork Message Structure.* In *Proceedings of the Sixth Data Communications Symposium,* IEEE. New York. November 1979.

[Postel80] Postel, J. B., *A Structured Format for Transmission of Multi-Media Documents.* RFC 767. Network Information Center, SRI International, Menlo Park, California. August 1980.

[Postel82] Postel, J. B., *Simple Mail Transfer Protocol.* RFC821 (obsoleting RFC788). Network Information Center, SRI International, Menlo Park,

California. August 1982.

[Schmidt79] Schmidt, E., *An Introduction to the Berkeley Network.* University of California, Berkeley California. 1979.

[Shoens79] Shoens, K., *Mail Reference Manual.* University of California, Berkeley. In UNIX Programmer's Manual, Seventh Edition, Volume 2C. December 1979.

[Sluizer81] Sluizer, S., and Postel, J. B., *Mail Transfer Protocol.* RFC 780. Network Information Center, SRI International, Menlo Park, California. May 1981.

[Solomon81] Solomon, M., Landweber, L., and Neuhengen, D., "The Design of the CSNET Name Server." CS-DN-2, University of Wisconsin, Madison. November 1981.

[Su82] Su, Zaw-Sing, and Postel, Jon, *The Domain Naming Convention for Internet User Applications.* RFC819. Network Information Center, SRI International, Menlo Park, California. August 1982.

[UNIX83] *The UNIX Programmer's Manual, Seventh Edition,* Virtual VAX-11 Version, Volume 1. Bell Laboratories, modified by the University of California, Berkeley, California. March, 1983.

UNIX Implementation

K. Thompson

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.

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

† UNIX is a trademark of Bell Laboratories.

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.

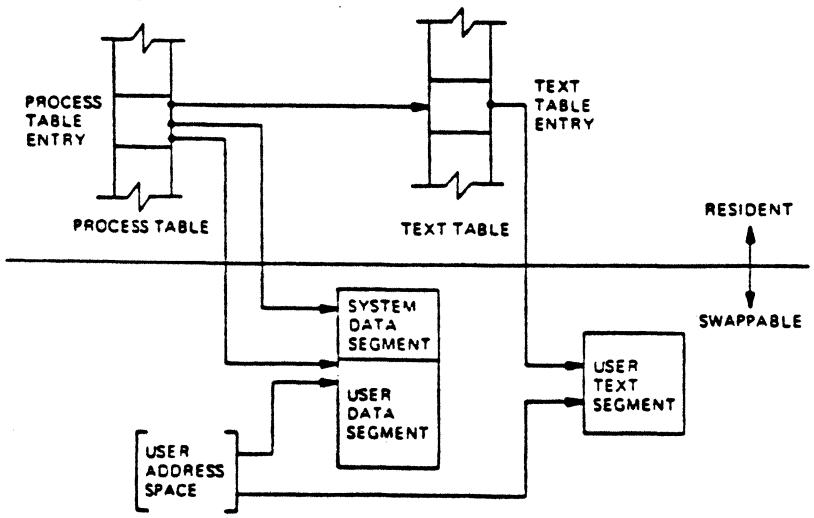


Fig. 1—Process control data structure.

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.

The user process accesses the file system with certain primitives. The most common of these are **open**, **create**, **read**, **write**, **seek**, and **close**. The data structures maintained are shown in Fig. 2.

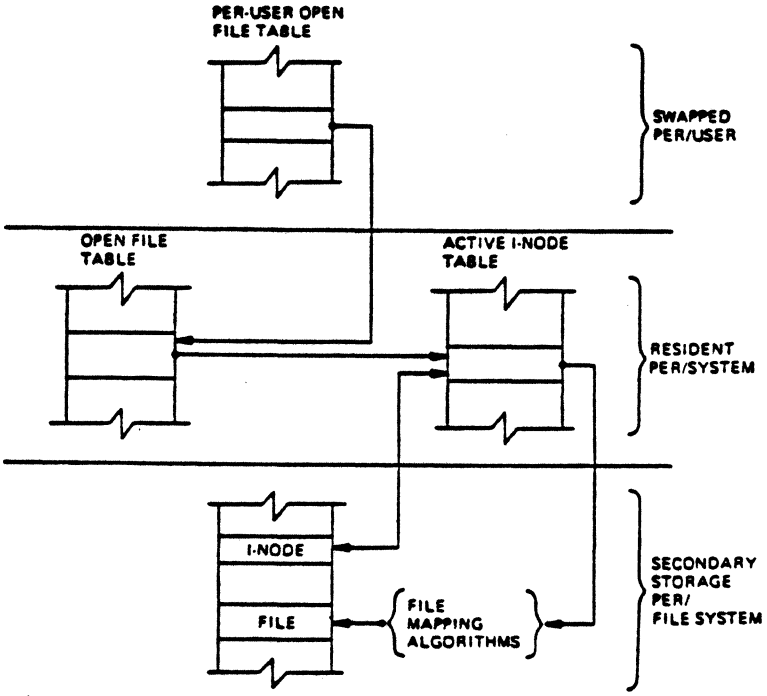


Fig. 2—File system data structure.

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

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

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

unnamed file vanishes when the file is closed. This is a method of obtaining temporary files.

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

4.2. Mounted file systems

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

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

4.3. Other system functions

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

The UNIX kernel is an I/O multiplexer more than a complete operating system. This is as it should be. Because of this outlook, many features are found in most other operating systems that are missing from the UNIX kernel. For example, the UNIX kernel does not support file access methods, file disposition, file formats, file maximum size, spooling, command language, logical records, physical records, assignment of logical file names, logical file names, more than one character set, an operator's console, an operator, log-in, or log-out. Many of these things are symptoms rather than features. Many of these things are implemented in user software using the kernel as a tool. A good example of this is the command language.[4] 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.[5]

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

The UNIX I/O System

Dennis M. Ritchie

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

†UNIX is a Trademark of Bell Laboratories.

terminated or closed the file (so the entry can be abandoned). There is also a 32-bit file offset which is used to indicate where in the file the next read or write will take place. Finally, there is a pointer to the entry for the file in the *inode* table, which contains a copy of the file's i-node.

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

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

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

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

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

Character Device Drivers

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

guarantees that events are unique.

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

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

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

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

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

The Block-device Interface

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

Seven routines constitute the most important part of the interface with the rest of the system. Given a device and block number, both ***bread*** and ***getblk*** return a pointer to a buffer header for the block; the difference is that ***bread*** is guaranteed to return a buffer actually containing the current data for the block, while ***getblk*** returns a buffer which contains the data in the block only if it is already in core (whether it is or not is indicated by the *B_DONE* bit; see below). In either case the buffer, and the corresponding device block, is made "busy," so that other processes referring to it are obliged to wait until it becomes free. *Getblk* is used, for example, when a block is

about to be totally rewritten, so that its previous contents are not useful; still, no other process can be allowed to refer to the block until the new data is placed into it.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

B_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.

On the Security of UNIX

Dennis M. Ritchie

Recently there has been much interest in the security aspects of operating systems and software. At issue is the ability to prevent undesired disclosure of information, destruction of information, and harm to the functioning of the system. This paper discusses the degree of security which can be provided under the UNIX† system and offers a number of hints on how to improve security.

The first fact to face is that UNIX was not developed with security, in any realistic sense, in mind; this fact alone guarantees a vast number of holes. (Actually the same statement can be made with respect to most systems.) The area of security in which UNIX is theoretically weakest is in protecting against crashing or at least crippling the operation of the system. The problem here is not mainly in uncritical acceptance of bad parameters to system calls— there may be bugs in this area, but none are known— but rather in lack of checks for excessive consumption of resources. Most notably, there is no limit on the amount of disk storage used, either in total space allocated or in the number of files or directories. Here is a particularly ghastly shell sequence guaranteed to stop the system:

```
while : ; do
        mkdir x
        cd x
done
```

Either a panic will occur because all the i-nodes on the device are used up, or all the disk blocks will be consumed, thus preventing anyone from writing files on the device.

In this version of the system, users are prevented from creating more than a set number of processes simultaneously, so unless users are in collusion it is unlikely that any one can stop the system altogether. However, creation of 20 or so CPU or disk-bound jobs leaves few resources available for others. Also, if many large jobs are run simultaneously, swap space may run out, causing a panic.

It should be evident that excessive consumption of disk space, files, swap space, and processes can easily occur accidentally in malfunctioning programs as well as at command level. In fact UNIX is essentially defenseless against this kind of abuse, nor is there any easy fix. The best that can be said is that it is generally fairly easy to detect what has happened when disaster strikes, to identify the user responsible, and take appropriate action. In practice, we have found that difficulties in this area are rather rare, but we have not been faced with malicious users, and enjoy a fairly generous supply of resources which have served to cushion us against accidental overconsumption.

The picture is considerably brighter in the area of protection of information from unauthorized perusal and destruction. Here the degree of security seems (almost) adequate theoretically, and the problems lie more in the necessity for care in the actual use of the system.

Each UNIX file has associated with it eleven bits of protection information together with a user identification number and a user-group identification number (UID and GID). Nine of the protection bits are used to specify independently permission to read, to write, and to execute the file to the user himself, to members of the user's group, and to all other users. Each process

† UNIX is a trademark of Bell Laboratories.

generated by or for a user has associated with it an effective UID and a real UID, and an effective and real GID. When an attempt is made to access the file for reading, writing, or execution, the user process's effective UID is compared against the file's UID; if a match is obtained, access is granted provided the read, write, or execute bit respectively for the user himself is present. If the UID for the file and for the process fail to match, but the GID's do match, the group bits are used; if the GID's do not match, the bits for other users are tested. The last two bits of each file's protection information, called the set-UID and set-GID bits, are used only when the file is executed as a program. If, in this case, the set-UID bit is on for the file, the effective UID for the process is changed to the UID associated with the file; the change persists until the process terminates or until the UID changed again by another execution of a set-UID file. Similarly the effective group ID of a process is changed to the GID associated with a file when that file is executed and has the set-GID bit set. The real UID and GID of a process do not change when any file is executed, but only as the result of a privileged system call.

The basic notion of the set-UID and set-GID bits is that one may write a program which is executable by others and which maintains files accessible to others only by that program. The classical example is the game-playing program which maintains records of the scores of its players. The program itself has to read and write the score file, but no one but the game's sponsor can be allowed unrestricted access to the file lest they manipulate the game to their own advantage. The solution is to turn on the set-UID bit of the game program. When, and only when, it is invoked by players of the game, it may update the score file but ordinary programs executed by others cannot access the score.

There are a number of special cases involved in determining access permissions. Since executing a directory as a program is a meaningless operation, the execute-permission bit, for directories, is taken instead to mean permission to search the directory for a given file during the scanning of a path name; thus if a directory has execute permission but no read permission for a given user, he may access files with known names in the directory, but may not read (list) the entire contents of the directory. Write permission on a directory is interpreted to mean that the user may create and delete files in that directory; it is impossible for any user to write directly into any directory.

Another, and from the point of view of security, much more serious special case is that there is a "super user" who is able to read any file and write any non-directory. The super-user is also able to change the protection mode and the owner UID and GID of any file and to invoke privileged system calls. It must be recognized that the mere notion of a super-user is a theoretical, and usually practical, blemish on any protection scheme.

The first necessity for a secure system is of course arranging that all files and directories have the proper protection modes. Traditionally, UNIX software has been exceedingly permissive in this regard; essentially all commands create files readable and writable by everyone. In the current version, this policy may be easily adjusted to suit the needs of the installation or the individual user. Associated with each process and its descendants is a mask, which is in effect *and*-ed with the mode of every file and directory created by that process. In this way, users can arrange that, by default, all their files are no more accessible than they wish. The standard mask, set by *login*, allows all permissions to the user himself and to his group, but disallows writing by others.

To maintain both data privacy and data integrity, it is necessary, and largely sufficient, to make one's files inaccessible to others. The lack of sufficiency could follow from the existence of set-UID programs created by the user and the possibility of total breach of system security in one of the ways discussed below (or one of the ways not discussed below). For greater protection, an encryption scheme is available. Since the editor is able to create encrypted documents, and the *crypt* command can be used to pipe such documents into the other text-processing programs, the length of time during which cleartext versions need be available is strictly limited. The encryption scheme used is not one of the strongest known, but it is judged adequate, in the sense that cryptanalysis is likely to require considerably more effort than more direct methods of reading the encrypted files. For example, a user who stores data that he regards as truly secret should be aware that he is implicitly trusting the system administrator not to install a version of the crypt command that stores every typed password in a file.

Needless to say, the system administrators must be at least as careful as their most demanding user to place the correct protection mode on the files under their control. In particular, it is necessary that special files be protected from writing, and probably reading, by ordinary users when they store sensitive files belonging to other users. It is easy to write programs that examine and change files by accessing the device on which the files live.

On the issue of password security, UNIX is probably better than most systems. Passwords are stored in an encrypted form which, in the absence of serious attention from specialists in the field, appears reasonably secure, provided its limitations are understood. In the current version, it is based on a slightly defective version of the Federal DES; it is purposely defective so that easily-available hardware is useless for attempts at exhaustive key-search. Since both the encryption algorithm and the encrypted passwords are available, exhaustive enumeration of potential passwords is still feasible up to a point. We have observed that users choose passwords that are easy to guess: they are short, or from a limited alphabet, or in a dictionary. Passwords should be at least six characters long and randomly chosen from an alphabet which includes digits and special characters.

Of course there also exist feasible non-cryptanalytic ways of finding out passwords. For example: write a program which types out "login: " on the typewriter and copies whatever is typed to a file of your own. Then invoke the command and go away until the victim arrives.

The set-UID (set-GID) notion must be used carefully if any security is to be maintained. The first thing to keep in mind is that a writable set-UID file can have another program copied onto it. For example, if the super-user *(su)* command is writable, anyone can copy the shell onto it and get a password-free version of *su*. A more subtle problem can come from set-UID programs which are not sufficiently careful of what is fed into them. To take an obsolete example, the previous version of the *mail* command was set-UID and owned by the super-user. This version sent mail to the recipient's own directory. The notion was that one should be able to send mail to anyone even if they want to protect their directories from writing. The trouble was that *mail* was rather dumb: anyone could mail someone else's private file to himself. Much more serious is the following scenario: make a file with a line like one in the password file which allows one to log in as the super-user. Then make a link named ".mail" to the password file in some writable directory on the same device as the password file (say /tmp). Finally mail the bogus login line to /tmp/.mail; You can then login as the super-user, clean up the incriminating evidence, and have your will.

The fact that users can mount their own disks and tapes as file systems can be another way of gaining super-user status. Once a disk pack is mounted, the system believes what is on it. Thus one can take a blank disk pack, put on it anything desired, and mount it. There are obvious and unfortunate consequences. For example: a mounted disk with garbage on it will crash the system; one of the files on the mounted disk can easily be a password-free version of *su;* other files can be unprotected entries for special files. The only easy fix for this problem is to forbid the use of *mount* to unprivileged users. A partial solution, not so restrictive, would be to have the *mount* command examine the special file for bad data, set-UID programs owned by others, and accessible special files, and balk at unprivileged invokers.

Password Security: A Case History Encryption Computing

Robert Morris

Ken Thompson

ABSTRACT

This paper describes the history of the design of the password security scheme on a remotely accessed time-sharing system. The present design was the result of countering observed attempts to penetrate the system. The result is a compromise between extreme security and ease of use.

April 3, 1978

Password Security: A Case History Encryption Computing

Robert Morris

Ken Thompson

INTRODUCTION

Password security on the UNIX† time-sharing system [1] is provided by a collection of programs whose elaborate and strange design is the outgrowth of many years of experience with earlier versions. To help develop a secure system, we have had a continuing competition to devise new ways to attack the security of the system (the bad guy) and, at the same time, to devise new techniques to resist the new attacks (the good guy). This competition has been in the same vein as the competition of long standing between manufacturers of armor plate and those of armor-piercing shells. For this reason, the description that follows will trace the history of the password system rather than simply presenting the program in its current state. In this way, the reasons for the design will be made clearer, as the design cannot be understood without also understanding the potential attacks.

An underlying goal has been to provide password security at minimal inconvenience to the users of the system. For example, those who want to run a completely open system without passwords, or to have passwords only at the option of the individual users, are able to do so, while those who require all of their users to have passwords gain a high degree of security against penetration of the system by unauthorized users.

The password system must be able not only to prevent any access to the system by unauthorized users (i.e. prevent them from logging in at all), but it must also prevent users who are already logged in from doing things that they are not authorized to do. The so called "super-user" password, for example, is especially critical because the super-user has all sorts of permissions and has essentially unlimited access to all system resources.

Password security is of course only one component of overall system security, but it is an essential component. Experience has shown that attempts to penetrate remote-access systems have been astonishingly sophisticated.

Remote-access systems are peculiarly vulnerable to penetration by outsiders as there are threats at the remote terminal, along the communications link, as well as at the computer itself. Although the security of a password encryption algorithm is an interesting intellectual and mathematical problem, it is only one tiny facet of a very large problem. In practice, physical security of the computer, communications security of the communications link, and physical control of the computer itself loom as far more important issues. Perhaps most important of all is control over the actions of ex-employees, since they are not under any direct control and they may have intimate knowledge about the system, its resources, and methods of access. Good system security involves realistic evaluation of the risks not only of deliberate attacks but also of casual unauthorized access and accidental disclosure.

PROLOGUE

The UNIX system was first implemented with a password file that contained the actual passwords of all the users, and for that reason the password file had to be heavily protected against being either read or written. Although historically, this had been the technique used for remote-access systems, it was completely unsatisfactory for several reasons.

† UNIX is a trademark of Bell Laboratories.

The technique is excessively vulnerable to lapses in security. Temporary loss of protection can occur when the password file is being edited or otherwise modified. There is no way to prevent the making of copies by privileged users. Experience with several earlier remote-access systems showed that such lapses occur with frightening frequency. Perhaps the most memorable such occasion occurred in the early 60's when a system administrator on the CTSS system at MIT was editing the password file and another system administrator was editing the daily message that is printed on everyone's terminal on login. Due to a software design error, the temporary editor files of the two users were interchanged and thus, for a time, the password file was printed on every terminal when it was logged in.

Once such a lapse in security has been discovered, everyone's password must be changed, usually simultaneously, at a considerable administrative cost. This is not a great matter, but far more serious is the high probability of such lapses going unnoticed by the system administrators.

Security against unauthorized disclosure of the passwords was, in the last analysis, impossible with this system because, for example, if the contents of the file system are put on to magnetic tape for backup, as they must be, then anyone who has physical access to the tape can read anything on it with no restriction.

Many programs must get information of various kinds about the users of the system, and these programs in general should have no special permission to read the password file. The information which should have been in the password file actually was distributed (or replicated) into a number of files, all of which had to be updated whenever a user was added to or dropped from the system.

THE FIRST SCHEME

The obvious solution is to arrange that the passwords not appear in the system at all, and it is not difficult to decide that this can be done by encrypting each user's password, putting only the encrypted form in the password file, and throwing away his original password (the one that he typed in). When the user later tries to log in to the system, the password that he types is encrypted and compared with the encrypted version in the password file. If the two match, his login attempt is accepted. Such a scheme was first described in [3, p.91ff.]. It also seemed advisable to devise a system in which neither the password file nor the password program itself needed to be protected against being read by anyone.

All that was needed to implement these ideas was to find a means of encryption that was very difficult to invert, even when the encryption program is available. Most of the standard encryption methods used (in the past) for encryption of messages are rather easy to invert. A convenient and rather good encryption program happened to exist on the system at the time; it simulated the M-209 cipher machine [4] used by the U.S. Army during World War II. It turned out that the M-209 program was usable, but with a given key, the ciphers produced by this program are trivial to invert. It is a much more difficult matter to find out the key given the cleartext input and the enciphered output of the program. Therefore, the password was used not as the text to be encrypted but as the key, and a constant was encrypted using this key. The encrypted result was entered into the password file.

ATTACKS ON THE FIRST APPROACH

Suppose that the bad guy has available the text of the password encryption program and the complete password file. Suppose also that he has substantial computing capacity at his disposal.

One obvious approach to penetrating the password mechanism is to attempt to find a general method of inverting the encryption algorithm. Very possibly this can be done, but few successful results have come to light, despite substantial efforts extending over a period of more than five years. The results have not proved to be very useful in penetrating systems.

Another approach to penetration is simply to keep trying potential passwords until one succeeds; this is a general cryptanalytic approach called ***key search***. Human beings being what they are, there is a strong tendency for people to choose relatively short and simple passwords that

they can remember. Given free choice, most people will choose their passwords from a restricted character set (e.g. all lower-case letters), and will often choose words or names. This human habit makes the key search job a great deal easier.

The critical factor involved in key search is the amount of time needed to encrypt a potential password and to check the result against an entry in the password file. The running time to encrypt one trial password and check the result turned out to be approximately 1.25 milliseconds on a PDP-11/70 when the encryption algorithm was recoded for maximum speed. It is takes essentially no more time to test the encrypted trial password against all the passwords in an entire password file, or for that matter, against any collection of encrypted passwords, perhaps collected from many installations.

If we want to check all passwords of length n that consist entirely of lower-case letters, the number of such passwords is 26^n. If we suppose that the password consists of printable characters only, then the number of possible passwords is somewhat less than 95^n. (The standard system "character erase" and "line kill" characters are, for example, not prime candidates.) We can immediately estimate the running time of a program that will test every password of a given length with all of its characters chosen from some set of characters. The following table gives estimates of the running time required on a PDP-11/70 to test all possible character strings of length n chosen from various sets of characters: namely, all lower-case letters, all lower-case letters plus digits, all alphanumeric characters, all 95 printable ASCII characters, and finally all 128 ASCII characters.

n	26 lower-case letters	36 lower-case letters and digits	62 alphanumeric characters	95 printable characters	all 128 ASCII characters
1	30 msec.	40 msec.	80 msec.	120 msec.	160 msec.
2	800 msec.	2 sec.	5 sec.	11 sec.	20 sec.
3	22 sec.	58 sec.	5 min.	17 min.	43 min.
4	10 min.	35 min.	5 hrs.	28 hrs.	93 hrs.
5	4 hrs.	21 hrs.	318 hrs.		
6	107 hrs.				

One has to conclude that it is no great matter for someone with access to a PDP-11 to test all lower-case alphabetic strings up to length five and, given access to the machine for, say, several weekends, to test all such strings up to six characters in length. By using such a program against a collection of actual encrypted passwords, a substantial fraction of all the passwords will be found.

Another profitable approach for the bad guy is to use the word list from a dictionary or to use a list of names. For example, a large commercial dictionary contains typicallly about 250,000 words; these words can be checked in about five minutes. Again, a noticeable fraction of any collection of passwords will be found. Improvements and extensions will be (and have been) found by a determined bad guy. Some "good" things to try are:

- The dictionary with the words spelled backwards.
- A list of first names (best obtained from some mailing list). Last names, street names, and city names also work well.
- The above with initial upper-case letters.
- All valid license plate numbers in your state. (This takes about five hours in New Jersey.)
- Room numbers, social security numbers, telephone numbers, and the like.

The authors have conducted experiments to try to determine typical users' habits in the choice of passwords when no constraint is put on their choice. The results were disappointing, except to the bad guy. In a collection of 3,289 passwords gathered from many users over a long period of time;

15 were a single ASCII character;

72 were strings of two ASCII characters;

464 were strings of three ASCII characters;

477 were string of four alphamerics;

706 were five letters, all upper-case or all lower-case;

605 were six letters, all lower-case.

An additional 492 passwords appeared in various available dictionaries, name lists, and the like. A total of 2,831, or 86% of this sample of passwords fell into one of these classes.

There was, of course, considerable overlap between the dictionary results and the character string searches. The dictionary search alone, which required only five minutes to run, produced about one third of the passwords.

Users could be urged (or forced) to use either longer passwords or passwords chosen from a larger character set, or the system could itself choose passwords for the users.

AN ANECDOTE

An entertaining and instructive example is the attempt made at one installation to force users to use less predictable passwords. The users did not choose their own passwords; the system supplied them. The supplied passwords were eight characters long and were taken from the character set consisting of lower-case letters and digits. They were generated by a pseudo-random number generator with only 2^{15} starting values. The time required to search (again on a PDP-11/70) through all character strings of length 8 from a 36-character alphabet is 112 years.

Unfortunately, only 2^{15} of them need be looked at, because that is the number of possible outputs of the random number generator. The bad guy did, in fact, generate and test each of these strings and found every one of the system-generated passwords using a total of only about one minute of machine time.

IMPROVEMENTS TO THE FIRST APPROACH

1. Slower Encryption

Obviously, the first algorithm used was far too fast. The announcement of the DES encryption algorithm [2] by the National Bureau of Standards was timely and fortunate. The DES is, by design, hard to invert, but equally valuable is the fact that it is extremely slow when implemented in software. The DES was implemented and used in the following way: The first eight characters of the user's password are used as a key for the DES; then the algorithm is used to encrypt a constant. Although this constant is zero at the moment, it is easily accessible and can be made installation-dependent. Then the DES algorithm is iterated 25 times and the resulting 64 bits are repacked to become a string of 11 printable characters.

2. Less Predictable Passwords

The password entry program was modified so as to urge the user to use more obscure passwords. If the user enters an alphabetic password (all upper-case or all lower-case) shorter than six characters, or a password from a larger character set shorter than five characters, then the program asks him to enter a longer password. This further reduces the efficacy of key search.

These improvements make it exceedingly difficult to find any individual password. The user is warned of the risks and if he cooperates, he is very safe indeed. On the other hand, he is not prevented from using his spouse's name if he wants to.

3. Salted Passwords

The key search technique is still likely to turn up a few passwords when it is used on a large collection of passwords, and it seemed wise to make this task as difficult as possible. To this end, when a password is first entered, the password program obtains a 12-bit random number (by reading the real-time clock) and appends this to the password typed in by the user. The concatenated

string is encrypted and both the 12-bit random quantity (called the *salt*) and the 64-bit result of the encryption are entered into the password file.

When the user later logs in to the system, the 12-bit quantity is extracted from the password file and appended to the typed password. The encrypted result is required, as before, to be the same as the remaining 64 bits in the password file. This modification does not increase the task of finding any individual password, starting from scratch, but now the work of testing a given character string against a large collection of encrypted passwords has been multiplied by 4096 (2^{12}). The reason for this is that there are 4096 encrypted versions of each password and one of them has been picked more or less at random by the system.

With this modification, it is likely that the bad guy can spend days of computer time trying to find a password on a system with hundreds of passwords, and find none at all. More important is the fact that it becomes impractical to prepare an encrypted dictionary in advance. Such an encrypted dictionary could be used to crack new passwords in milliseconds when they appear.

There is a (not inadvertent) side effect of this modification. It becomes nearly impossible to find out whether a person with passwords on two or more systems has used the same password on all of them, unless you already know that.

4. The Threat of the DES Chip

Chips to perform the DES encryption are already commercially available and they are very fast. The use of such a chip speeds up the process of password hunting by three orders of magnitude. To avert this possibility, one of the internal tables of the DES algorithm (in particular, the so-called E-table) is changed in a way that depends on the 12-bit random number. The E-table is inseparably wired into the DES chip, so that the commercial chip cannot be used. Obviously, the bad guy could have his own chip designed and built, but the cost would be unthinkable.

5. A Subtle Point

To login successfully on the UNIX system, it is necessary after dialing in to type a valid user name, and then the correct password for that user name. It is poor design to write the login command in such a way that it tells an interloper when he has typed in a invalid user name. The response to an invalid name should be identical to that for a valid name.

When the slow encryption algorithm was first implemented, the encryption was done only if the user name was valid, because otherwise there was no encrypted password to compare with the supplied password. The result was that the response was delayed by about one-half second if the name was valid, but was immediate if invalid. The bad guy could find out whether a particular user name was valid. The routine was modified to do the encryption in either case.

CONCLUSIONS

On the issue of password security, UNIX is probably better than most systems. The use of encrypted passwords appears reasonably secure in the absence of serious attention of experts in the field.

It is also worth some effort to conceal even the encrypted passwords. Some UNIX systems have instituted what is called an "external security code" that must be typed when dialing into the system, but before logging in. If this code is changed periodically, then someone with an old password will likely be prevented from using it.

Whenever any security procedure is instituted that attempts to deny access to unauthorized persons, it is wise to keep a record of both successful and unsuccessful attempts to get at the secured resource. Just as an out-of-hours visitor to a computer center normally must not only identify himself, but a record is usually also kept of his entry. Just so, it is a wise precaution to make and keep a record of all attempts to log into a remote-access time-sharing system, and certainly all unsuccessful attempts.

Bad guys fall on a spectrum whose one end is someone with ordinary access to a system and whose goal is to find out a particular password (usually that of the super-user) and, at the other

end, someone who wishes to collect as much password information as possible from as many systems as possible. Most of the work reported here serves to frustrate the latter type; our experience indicates that the former type of bad guy never was very successful.

We recognize that a time-sharing system must operate in a hostile environment. We did not attempt to hide the security aspects of the operating system, thereby playing the customary make-believe game in which weaknesses of the system are not discussed no matter how apparent. Rather we advertised the password algorithm and invited attack in the belief that this approach would minimize future trouble. The approach has been successful.

References

[1] Ritchie, D.M. and Thompson, K. The UNIX Time-Sharing System. *Comm. ACM* **17** (July 1974), pp. 365-375.

[2] *Proposed Federal Information Processing Data Encryption Standard.* Federal Register (40FR12134), March 17, 1975

[3] Wilkes, M. V. *Time-Sharing Computer Systems.* American Elsevier, New York, (1968).

[4] U. S. Patent Number 2,089,603.

Uucp Implementation Description

D. A. Nowitz

Introduction

Uucp is a series of programs designed to permit communication between UNIX† systems using either dial-up or hardwired communication lines. It is used for file transfers and remote command execution. The first version of the system was designed and implemented by M. E. Lesk.[1] This paper describes the current (second) implementation of the system.

Uucp is a batch type operation. Files are created in a spool directory for processing by the uucp demons. There are three types of files used for the execution of work. *Data files* contain data for transfer to remote systems. *Work files* contain directions for file transfers between systems. *Execution files* are directions for UNIX command executions which involve the resources of one or more systems.

The uucp system consists of four primary and two secondary programs. The primary programs are:

uucp	This program creates work and gathers data files in the spool directory for the transmission of files.
uux	This program creates work files, execute files and gathers data files for the remote execution of UNIX commands.
uucico	This program executes the work files for data transmission.
uuxqt	This program executes the execution files for UNIX command execution.

The secondary programs are:

uulog	This program updates the log file with new entries and reports on the status of uucp requests.
uuclean	This program removes old files from the spool directory.

The remainder of this paper will describe the operation of each program, the installation of the system, the security aspects of the system, the files required for execution, and the administration of the system.

1. Uucp - UNIX to UNIX File Copy

The *uucp* command is the user's primary interface with the system. The *uucp* command was designed to look like *cp* to the user. The syntax is

uucp [option] ... source ... destination

where the source and destination may contain the prefix *system-name!* which indicates the system on which the file or files reside or where they will be copied.

The options interpreted by *uucp* are:

-d	Make directories when necessary for copying the file.
-c	Don't copy source files to the spool directory, but use the specified source when the actual transfer takes place.

† UNIX is a trademark of Bell Laboratories.

1 M. E. Lesk and A. S. Cohen, UNIX Software Distribution by Communication Link, private communication.

-g*letter* Put *letter* in as the grade in the name of the work file. (This can be used to change the order of work for a particular machine.)

-m Send mail on completion of the work.

The following options are used primarily for debugging:

-r Queue the job but do not start *uucico* program.

-s*dir* Use directory *dir* for the spool directory.

-x*num* *Num* is the level of debugging output desired.

The destination may be a directory name, in which case the file name is taken from the last part of the source's name. The source name may contain special shell characters such as "*?*//*". If a source argument has a *system-name!* prefix for a remote system, the file name expansion will be done on the remote system.

The command

uucp *.c usg!/usr/dan

will set up the transfer of all files whose names end with ".c" to the "/usr/dan" directory on the"usg" machine.

The source and/or destination names may also contain a ~*user* prefix. This translates to the login directory on the specified system. For names with partial path-names, the current directory is prepended to the file name. File names with ../ are not permitted.

The command

uucp usg!~dan/*.h ~dan

will set up the transfer of files whose names end with ".h" in dan's login directory on system "usg" to dan's local login directory.

For each source file, the program will check the source and destination file-names and the system-part of each to classify the work into one of five types:

[1] Copy source to destination on local system.

[2] Receive files from other systems.

[3] Send files to a remote systems.

[4] Send files from remote systems to another remote system.

[5] Receive files from remote systems when the source contains special shell characters as mentioned above.

After the work has been set up in the spool directory, the *uucico* program is started to try to contact the other machine to execute the work (unless the -r option was specified).

Type 1

A *cp* command is used to do the work. The *-d* and the *-m* options are not honored in this case.

Type 2

A one line *work file* is created for each file requested and put in the spool directory with the following fields, each separated by a blank. (All *work files* and *execute files* use a blank as the field separator.)

[1] R

[2] The full path-name of the source or a ~user/path-name. The ~*user* part will be expanded on the remote system.

[3] The full path-name of the destination file. If the ~*user* notation is used, it will be immediately expanded to be the login directory for the user.

[4] The user's login name.

[5] A "–" followed by an option list. (Only the –m and –d options will appear in this list.)

Type 3

For each source file, a ***work file*** is created and the source file is copied into a ***data file*** in the spool directory. (A "–c" option on the ***uucp*** command will prevent the ***data file*** from being made.) In this case, the file will be transmitted from the indicated source.) The fields of each entry are given below.

[1] S

[2] The full-path name of the source file.

[3] The full-path name of the destination or ~user/file-name.

[4] The user's login name.

[5] A "–" followed by an option list.

[6] The name of the ***data file*** in the spool directory.

[7] The file mode bits of the source file in octal print format (e.g. 0666).

Type 4 and Type 5

Uucp generates a ***uucp*** command and sends it to the remote machine; the remote ***uucico*** executes the ***uucp*** command.

2. Uux - UNIX To UNIX Execution

The ***uux*** command is used to set up the execution of a UNIX command where the execution machine and/or some of the files are remote. The syntax of the uux command is

uux [–] [option] ... command-string

where the command-string is made up of one or more arguments. All special shell characters such as "< > | ^" must be quoted either by quoting the entire command-string or quoting the character as a separate argument. Within the command-string, the command and file names may contain a ***system-name!*** prefix. All arguments which do not contain a "!" will not be treated as files. (They will not be copied to the execution machine.) The "–" is used to indicate that the standard input for ***command-string*** should be inherited from the standard input of the ***uux*** command. The options, essentially for debugging, are:

–r Don't start ***uucico*** or ***uuxqt*** after queuing the job;

–x***num*** Num is the level of debugging output desired.

The command

pr abc | uux – usg!lpr

will set up the output of "pr abc" as standard input to an lpr command to be executed on system "usg".

Uux generates an ***execute file*** which contains the names of the files required for execution (including standard input), the user's login name, the destination of the standard output, and the command to be executed. This file is either put in the spool directory for local execution or sent to the remote system using a generated send command (type 3 above).

For required files which are not on the execution machine, ***uux*** will generate receive command files (type 2 above). These command-files will be put on the execution machine and executed by the ***uucico*** program. (This will work only if the local system has permission to put files in the remote spool directory as controlled by the remote *USERFILE*.)

The ***execute file*** will be processed by the ***uuxqt*** program on the execution machine. It is made up of several lines, each of which contains an identification character and one or more arguments. The order of the lines in the file is not relevant and some of the lines may not be present. Each line is described below.

User Line

U user system

where the *user* and *system* are the requester's login name and system.

Required File Line

F file-name real-name

where the *file-name* is the generated name of a file for the execute machine and *real-name* is the last part of the actual file name (contains no path information). Zero or more of these lines may be present in the *execute file*. The *uuxqt* program will check for the existence of all required files before the command is executed.

Standard Input Line

I file-name

The standard input is either specified by a "<" in the command-string or inherited from the standard input of the *uux* command if the "-" option is used. If a standard input is not specified, "/dev/null" is used.

Standard Output Line

O file-name system-name

The standard output is specified by a ">" within the command-string. If a standard output is not specified, "/dev/null" is used. (Note – the use of ">>" is not implemented.)

Command Line

C command [arguments] ...

The arguments are those specified in the command-string. The standard input and standard output will not appear on this line. All *required files* will be moved to the execution directory (a subdirectory of the spool directory) and the UNIX command is executed using the Shell specified in the *uucp.h* header file. In addition, a shell "PATH" statement is prepended to the command line as specified in the *uuxqt* program.

After execution, the standard output is copied or set up to be sent to the proper place.

3. Uucico - Copy In, Copy Out

The *uucico* program will perform the following major functions:

- Scan the spool directory for work.
- Place a call to a remote system.
- Negotiate a line protocol to be used.
- Execute all requests from both systems.
- Log work requests and work completions.

Uucico may be started in several ways;

a) by a system daemon,

b) by one of the *uucp, uux, uuxqt* or *uucico* programs,

c) directly by the user (this is usually for testing),

d) by a remote system. (The uucico program should be specified as the "shell" field in the "/etc/passwd" file for the "uucp" logins.)

When started by method a, b or c, the program is considered to be in *MASTER* mode. In this mode, a connection will be made to a remote system. If started by a remote system (method d), the program is considered to be in *SLAVE* mode.

The *MASTER* mode will operate in one of two ways. If no system name is specified (–s option not specified) the program will scan the spool directory for systems to call. If a system name is specified, that system will be called, and work will only be done for that system.

The *uucico* program is generally started by another program. There are several options used for execution:

–r1 Start the program in *MASTER* mode. This is used when *uucico* is started by a program or "cron" shell.

–s*sys* Do work only for system *sys*. If *–s* is specified, a call to the specified system will be made even if there is no work for system *sys* in the spool directory. This is useful for polling systems which do not have the hardware to initiate a connection.

The following options are used primarily for debugging:

–d*dir* Use directory *dir* for the spool directory.

–x*num* *Num* is the level of debugging output desired.

The next part of this section will describe the major steps within the *uucico* program.

Scan For Work

The names of the work related files in the spool directory have format

type . system-name grade number

where:

Type is an upper case letter, (*C* - copy command file, *D* - data file, *X* - execute file);

System-name is the remote system;

Grade is a character;

Number is a four digit, padded sequence number.

The file

C.res45n0031

would be a *work file* for a file transfer between the local machine and the "res45" machine.

The scan for work is done by looking through the spool directory for *work files* (files with prefix "C."). A list is made of all systems to be called. *Uucico* will then call each system and process all *work files*.

Call Remote System

The call is made using information from several files which reside in the uucp program directory. At the start of the call process, a lock is set to forbid multiple conversations between the same two systems.

The system name is found in the *L.sys* file. The information contained for each system is;

[1] system name,

[2] times to call the system (days-of-week and times-of-day),

[3] device or device type to be used for call,

[4] line speed,

[5] phone number if field [3] is *ACU* or the device name (same as field [3]) if not *ACU,*

[6] login information (multiple fields),

The time field is checked against the present time to see if the call should be made.

The *phone number* may contain abbreviations (e.g. mh, py, boston) which get translated into dial sequences using the *L-dialcodes* file.

The *L-devices* file is scanned using fields [3] and [4] from the *L.sys* file to find an available device for the call. The program will try all devices which satisfy [3] and [4] until the call is made, or no more devices can be tried. If a device is successfully opened, a lock file is created so that another copy of *uucico* will not try to use it. If the call is complete, the *login information* (field [6] of *L.sys*) is used to login.

The conversation between the two *uucico* programs begins with a handshake started by the called, *SLAVE*, system. The *SLAVE* sends a message to let the *MASTER* know it is ready to receive the system identification and conversation sequence number. The response from the *MASTER* is verified by the *SLAVE* and if acceptable, protocol selection begins. The *SLAVE* can also reply with a "call-back required" message in which case, the current conversation is terminated.

Line Protocol Selection

The remote system sends a message

P*proto-list*

where proto-list is a string of characters, each representing a line protocol.

The calling program checks the proto-list for a letter corresponding to an available line protocol and returns a *use-protocol* message. The *use-protocol* message is

U*code*

where code is either a one character protocol letter or *N* which means there is no common protocol.

Work Processing

The initial roles (*MASTER* or *SLAVE*) for the work processing are the mode in which each program starts. (The *MASTER* has been specified by the "-r1" uucico option.) The *MASTER* program does a work search similar to the one used in the "Scan For Work" section.

There are five messages used during the work processing, each specified by the first character of the message. They are;

S send a file,

R receive a file,

C copy complete,

X execute a *uucp* command,

H hangup.

The *MASTER* will send *R*, *S* or *X* messages until all work from the spool directory is complete, at which point an *H* message will be sent. The *SLAVE* will reply with *SY, SN, RY, RN, HY, HN, XY, XN, corresponding to yes or no for each request.*

The send and receive replies are based on permission to access the requested file/directory using the *USERFILE* and read/write permissions of the file/directory. After each file is copied into the spool directory of the receiving system, a copy-complete message is sent by the receiver of the file. The message *CY* will be sent if the file has successfully been moved from the temporary spool file to the actual destination. Otherwise, a *CN* message is sent. (In the case of *CN*, the transferred file will be in the spool directory with a name beginning with "TM'.) The requests and results are logged on both systems.

The hangup response is determined by the *SLAVE* program by a work scan of the spool directory. If work for the remote system exists in the *SLAVE's* spool directory, an *HN* message is sent and the programs switch roles. If no work exists, an *HY* response is sent.

Conversation Termination

When a *HY* message is received by the *MASTER* it is echoed back to the *SLAVE* and the protocols are turned off. Each program sends a final "OO" message to the other. The original *SLAVE* program will clean up and terminate. The *MASTER* will proceed to call other systems and process work as long as possible or terminate if a *-s* option was specified.

4. Uuxqt - Uucp Command Execution

The *uuxqt* program is used to execute *execute files* generated by *uux*. The *uuxqt* program may be started by either the *uucico* or *uux* programs. The program scans the spool directory for *execute files* (prefix "X."). Each one is checked to see if all the required files are available and if so, the command line or send line is executed.

The *execute file* is described in the "Uux" section above.

Command Execution

The execution is accomplished by executing a *sh -c* of the command line after appropriate standard input and standard output have been opened. If a standard output is specified, the program will create a send command or copy the output file as appropriate.

5. Uulog - Uucp Log Inquiry

The *uucp* programs create individual log files for each program invocation. Periodically, *uulog* may be executed to prepend these files to the system logfile. This method of logging was chosen to minimize file locking of the logfile during program execution.

The *uulog* program merges the individual log files and outputs specified log entries. The output request is specified by the use of the following options:

-s*sys*	Print entries where *sys* is the remote system name;
-u*user*	Print entries for user *user*.

The intersection of lines satisfying the two options is output. A null *sys* or *user* means all system names or users respectively.

6. Uuclean - Uucp Spool Directory Cleanup

This program is typically started by the daemon, once a day. Its function is to remove files from the spool directory which are more than 3 days old. These are usually files for work which can not be completed.

The options available are:

-d*dir*	The directory to be scanned is *dir*.
-m	Send mail to the owner of each file being removed. (Note that most files put into the spool directory will be owned by the owner of the uucp programs since the setuid bit will be set on these programs. The mail will therefore most often go to the owner of the uucp programs.)
-n*hours*	Change the aging time from 72 hours to *hours* hours.
-p*pre*	Examine files with prefix *pre* for deletion. (Up to 10 file prefixes may be specified.)
-x*num*	This is the level of debugging output desired.

7. Security

The uucp system, left unrestricted, will let any outside user execute any commands and copy in/out any file which is readable/writable by the uucp login user. It is up to the individual sites to be aware of this and

apply the protections that they feel are necessary.

There are several security features available aside from the normal file mode protections. These must be set up by the installer of the *uucp* system.

- The login for uucp does not get a standard shell. Instead, the *uucico* program is started. Therefore, the only work that can be done is through *uucico*.
- A path check is done on file names that are to be sent or received. The *USERFILE* supplies the information for these checks. The *USERFILE* can also be set up to require call-back for certain login-ids. (See the "Files required for execution" section for the file description.)
- A conversation sequence count can be set up so that the called system can be more confident that the caller is who he says he is.
- The *uuxqt* program comes with a list of commands that it will execute. A "PATH" shell statement is prepended to the command line as specifed in the *uuxqt* program. The installer may modify the list or remove the restrictions as desired.
- The *L.sys* file should be owned by uucp and have mode 0400 to protect the phone numbers and login information for remote sites. (Programs uucp, uucico, uux, uuxqt should be also owned by uucp and have the setuid bit set.)

8. Uucp Installation

There are several source modifications that may be required before the system programs are compiled. These relate to the directories used during compilation, the directories used during execution, and the local *uucp system-name*.

The four directories are:

lib	(/usr/src/cmd/uucp) This directory contains the source files for generating the *uucp* system.
program	(/usr/lib/uucp) This is the directory used for the executable system programs and the system files.
spool	(/usr/spool/uucp) This is the spool directory used during *uucp* execution.
xqtdir	(/usr/spool/uucp/.XQTDIR) This directory is used during execution of *execute files*.

The names given in parentheses above are the default values for the directories. The italicized named *lib, program, xqtdir,* and *spool* will be used in the following text to represent the appropriate directory names.

There are two files which may require modification, the *makefile* file and the *uucp.h* file. The following paragraphs describe the modifications. The modes of *spool* and *xqtdir* should be made "0777".

Uucp.h modification

Change the *program* and the *spool* names from the default values to the directory names to be used on the local system using global edit commands.

Change the *define* value for *MYNAME* to be the local *uucp* system-name.

makefile modification

There are several *make* variable definitions which may need modification.

INSDIR	This is the *program* directory (e.g. INSDIR=/usr/lib/uucp). This parameter is used if "make cp" is used after the programs are compiled.
IOCTL	This is required to be set if an appropriate *ioctl* interface subroutine does not exist in the standard "C" library; the statement "IOCTL=ioctl.o" is required in this case.

PKON The statement "PKON=pkon.o" is required if the packet driver is not in the kernel.

Compile the system The command

make

will compile the entire system. The command

make cp

will copy the commands to the to the appropriate directories.

The programs ***uucp***, ***uux***, and ***uulog*** should be put in "/usr/bin". The programs ***uuxqt***, ***uucico***, and ***uuclean*** should be put in the ***program*** directory.

Files required for execution

There are four files which are required for execution, all of which should reside in the *program* directory. The field separator for all files is a space unless otherwise specified.

L-devices

This file contains entries for the call-unit devices and hardwired connections which are to be used by *uucp*. The special device files are assumed to be in the */dev* directory. The format for each entry is

line call-unit speed

where;

line is the device for the line (e.g. cul0),

call-unit is the automatic call unit associated with *line* (e.g. cua0), (Hardwired lines have a number "0" in this field.),

speed is the line speed.

The line

cul0 cua0 300

would be set up for a system which had device cul0 wired to a call-unit cua0 for use at 300 baud.

L-dialcodes

This file contains entries with location abbreviations used in the *L.sys* file (e.g. py, mh, boston). The entry format is

abb dial-seq

where;

abb is the abbreviation,

dial-seq is the dial sequence to call that location.

The line

py 165–

would be set up so that entry py7777 would send 165–7777 to the dial-unit.

LOGIN/SYSTEM NAMES

It is assumed that the ***login name*** used by a remote computer to call into a local computer is not the same as the login name of a normal user of that local machine. However, several remote computers may employ the same login name.

Each computer is given a unique ***system name*** which is transmitted at the start of each call. This name identifies the calling machine to the called machine.

USERFILE

This file contains user accessibility information. It specifies four types of constraint;

[1] which files can be accessed by a normal user of the local machine,

[2] which files can be accessed from a remote computer,

[3] which login name is used by a particular remote computer,

[4] whether a remote computer should be called back in order to confirm its identity.

Each line in the file has the following format

login,sys [c] path-name [path-name] ...

where;

login	is the login name for a user or the remote computer,
sys	is the system name for a remote computer,
c	is the optional ***call-back required*** flag,
path-name	is a path-name prefix that is acceptable for ***user***.

The constraints are implemented as follows.

[1] When the program is obeying a command stored on the local machine, *MASTER* mode, the path-names allowed are those given for the first line in the *USERFILE* that has a login name that matches the login name of the user who entered the command. If no such line is found, the first line with a ***null*** login name is used.

[2] When the program is responding to a command from a remote machine, *SLAVE* mode, the path-names allowed are those given for the first line in the file that has the system name that matches the system name of the remote machine. If no such line is found, the first one with a ***null*** system name is used.

[3] When a remote computer logs in, the login name that it uses must appear in the *USERFILE*. There may be several lines with the same login name but one of them must either have the name of the remote system or must contain a ***null*** system name.

[4] If the line matched in ([3]) contains a "c", the remote machine is called back before any transactions take place.

The line

```
u,m  /usr/xyz
```

allows machine ***m*** to login with name ***u*** and request the transfer of files whose names start with "/usr/xyz".

The line

```
dan,  /usr/dan
```

allows the ordinary user ***dan*** to issue commands for files whose name starts with "/usr/dan".

The lines

```
u,m /usr/xyz  /usr/spool
u,  /usr/spool
```

allows any remote machine to login with name ***u***, but if its system name is not ***m***, it can only ask to transfer files whose names start with "/usr/spool".

The lines

```
root,  /
,  /usr
```

allows any user to transfer files beginning with "/usr" but the user with login ***root*** can transfer any file.

L.sys

Each entry in this file represents one system which can be called by the local uucp programs. The fields are described below.

system name

The name of the remote system.

time

This is a string which indicates the days-of-week and times-of-day when the system should be called (e.g. MoTuTh0800–1730).

The day portion may be a list containing some of

Su Mo Tu We Th Fr Sa

or it may be *Wk* for any week-day or *Any* for any day.

The time should be a range of times (e.g. 0800–1230). If no time portion is specified, any time of day is assumed to be ok for the call.

device

This is either *ACU* or the hardwired device to be used for the call. For the hardwired case, the last part of the special file name is used (e.g. tty0).

speed

This is the line speed for the call (e.g. 300).

phone

The phone number is made up of an optional alphabetic abbreviation and a numeric part. The abbreviation is one which appears in the *L-dialcodes* file (e.g. mh5900, boston995–9980).

For the hardwired devices, this field contains the same string as used for the *device* field.

login

The login information is given as a series of fields and subfields in the format

expect send [expect send] ...

where; *expect* is the string expected to be read and *send* is the string to be sent when the *expect* string is received.

The expect field may be made up of subfields of the form

expect[–send–expect]...

where the *send* is sent if the prior *expect* is not successfully read and the *expect* following the *send* is the next expected string.

There are two special names available to be sent during the login sequence. The string *EOT* will send an EOT character and the string *BREAK* will try to send a BREAK character. (The *BREAK* character is simulated using line speed changes and null characters and may not work on all devices and/or systems.)

A typical entry in the L.sys file would be

sys Any ACU 300 mh7654 login uucp ssword: word

The expect algorithm looks at the last part of the string as illustrated in the password field.

9. Administration

This section indicates some events and files which must be administered for the *uucp* system. Some administration can be accomplished by *shell files* which can be initiated by *crontab* entries. Others will require manual intervention. Some sample *shell files* are given toward the end of this section.

SQFILE – sequence check file

This file is set up in the *program* directory and contains an entry for each remote system with which you agree to perform conversation sequence checks. The initial entry is just the system name of the remote system. The first conversation will add two items to the line, the conversation count, and the date/time of the most resent conversation. These items will be updated with each conversation. If a sequence check fails, the entry will have to be adjusted.

TM – temporary data files

These files are created in the *spool* directory while files are being copied from a remote machine. Their names have the form

TM.pid.ddd

where *pid* is a process-id and *ddd* is a sequential three digit number starting at zero for each invocation of *uucico* and incremented for each file received.

After the entire remote file is received, the *TM* file is moved/copied to the requested destination. If processing is abnormally terminated or the move/copy fails, the file will remain in the spool directory.

The leftover files should be periodically removed; the *uuclean* program is useful in this regard. The command

uuclean –pTM

will remove all *TM* files older than three days.

LOG – log entry files

During execution of programs, individual *LOG* files are created in the *spool* directory with information about queued requests, calls to remote systems, execution of *uux* commands and file copy results. These files should be combined into the *LOGFILE* by using the *uulog* program. This program will put the new *LOG* files at the beginning of the existing *LOGFILE*. The command

uulog

will accomplish the merge. Options are available to print some or all the log entries after the files are merged. The *LOGFILE* should be removed periodically since it is copied each time new LOG entries are put into the file.

The *LOG* files are created initially with mode 0222. If the program which creates the file terminates normally, it changes the mode to 0666. Aborted runs may leave the files with mode 0222 and the *uulog* program will not read or remove them. To remove them, either use *rm*, *uuclean*, or change the mode to 0666 and let *uulog* merge them with the *LOGFILE*.

STST – system status files

These files are created in the spool directory by the *uucico* program. They contain information of failures such as login, dialup or sequence check and will contain a *TALKING* status when to machines are conversing. The form of the file name is

STST.sys

where *sys* is the remote system name.

For ordinary failures (dialup, login), the file will prevent repeated tries for about one hour. For sequence check failures, the file must be removed before any future attempts to converse with that

remote system.

If the file is left due to an aborted run, it may contain a *TALKING* status. In this case, the file must be removed before a conversation is attempted.

LCK – lock files

Lock files are created for each device in use (e.g. automatic calling unit) and each system conversing. This prevents duplicate conversations and multiple attempts to use the same devices. The form of the lock file name is

LCK..str

where *str* is either a device or system name. The files may be left in the spool directory if runs abort. They will be ignored (reused) after a time of about 24 hours. When runs abort and calls are desired before the time limit, the lock files should be removed.

Shell Files

The *uucp* program will spool work and attempt to start the *uucico* program, but the starting of *uucico* will sometimes fail. (No devices available, login failures etc.). Therefore, the *uucico* program should be periodically started. The command to start *uucico* can be put in a "shell" file with a command to merge *LOG* files and started by a crontab entry on an hourly basis. The file could contain the commands

program /uulog
program /uucico –r1

Note that the "–r1" option is required to start the *uucico* program in *MASTER* mode.

Another shell file may be set up on a daily basis to remove *TM*, *ST* and *LCK* files and *C.* or *D.* files for work which can not be accomplished for reasons like bad phone number, login changes etc. A shell file containing commands like

program /uuclean –pTM –pC. –pD.
program /uuclean –pST –pLCK –n12

can be used. Note the "–n12" option causes the *ST* and *LCK* files older than 12 hours to be deleted. The absence of the "–n" option will use a three day time limit.

A daily or weekly shell should also be created to remove or save old *LOGFILE*s. A shell like

cp *spool* /LOGFILE *spool* /o.LOGFILE
rm *spool* /LOGFILE

can be used.

Login Entry

One or more logins should be set up for *uucp*. Each of the "/etc/passwd" entries should have the "*program*/uucico" as the shell to be executed. The login directory is not used, but if the system has a special directory for use by the users for sending or receiving file, it should as the login entry. The various logins are used in conjunction with the *USERFILE* to restrict file access. Specifying the *shell* argument limits the login to the use of uucp (*uucico*) only.

File Modes

It is suggested that the owner and file modes of various programs and files be set as follows.

The programs *uucp*, *uux*, *uucico* and *uuxqt* should be owned by the *uucp* login with the "setuid" bit set and only execute permissions (e.g. mode 04111). This will prevent outsiders from modifying the programs to get at a standard *shell* for the *uucp* logins.

The *L.sys*, *SQFILE* and the *USERFILE* which are put in the *program* directory should be owned by the *uucp* login and set with mode 0400.

Uucp Implementation Description

D. A. Nowitz

ABSTRACT

Uucp is a series of programs designed to permit communication between UNIX systems using either dial-up or hardwired communication lines. This document gives a detailed implementation description of the current (second) implementation of uucp.

This document is for use by an administrator/installer of the system. It is not meant as a user's guide.

October 31, 1978

A Dial-Up Network of UNIX™ Systems

D. A. Nowitz

M. E. Lesk

ABSTRACT

A network of over eighty UNIX† computer systems has been established using the telephone system as its primary communication medium. The network was designed to meet the growing demands for software distribution and exchange. Some advantages of our design are:

- The startup cost is low. A system needs only a dial-up port, but systems with automatic calling units have much more flexibility.
- No operating system changes are required to install or use the system.
- The communication is basically over dial-up lines, however, hardwired communication lines can be used to increase speed.
- The command for sending/receiving files is simple to use.

Keywords: networks, communications, software distribution, software maintenance

August 18, 1978

† UNIX is a trademark of Bell Laboratories.

A Dial-Up Network of UNIX™ Systems

D. A. Nowitz

M. E. Lesk

1. Purpose

The widespread use of the UNIX system ritchie thompson bstj 1978 within Bell Laboratories has produced problems of software distribution and maintenance. A conventional mechanism was set up to distribute the operating system and associated programs from a central site to the various users. However this mechanism alone does not meet all software distribution needs. Remote sites generate much software and must transmit it to other sites. Some UNIX systems are themselves central sites for redistribution of a particular specialized utility, such as the Switching Control Center System. Other sites have particular, often long-distance needs for software exchange; switching research, for example, is carried on in New Jersey, Illinois, Ohio, and Colorado. In addition, general purpose utility programs are written at all UNIX system sites. The UNIX system is modified and enhanced by many people in many places and it would be very constricting to deliver new software in a one-way stream without any alternative for the user sites to respond with changes of their own.

Straightforward software distribution is only part of the problem. A large project may exceed the capacity of a single computer and several machines may be used by the one group of people. It then becomes necessary for them to pass messages, data and other information back an forth between computers.

Several groups with similar problems, both inside and outside of Bell Laboratories, have constructed networks built of hardwired connections only. dolotta mashey 1978 bstj network unix system chesson Our network, however, uses both dial-up and hardwired connections so that service can be provided to as many sites as possible.

2. Design Goals

Although some of our machines are connected directly, others can only communicate over low-speed dial-up lines. Since the dial-up lines are often unavailable and file transfers may take considerable time, we spool all work and transmit in the background. We also had to adapt to a community of systems which are independently operated and resistant to suggestions that they should all buy particular hardware or install particular operating system modifications. Therefore, we make minimal demands on the local sites in the network. Our implementation requires no operating system changes; in fact, the transfer programs look like any other user entering the system through the normal dial-up login ports, and obeying all local protection rules.

We distinguish "active" and "passive" systems on the network. Active systems have an automatic calling unit or a hardwired line to another system, and can initiate a connection. Passive systems do not have the hardware to initiate a connection. However, an active system can be assigned the job of calling passive systems and executing work found there; this makes a passive system the functional equivalent of an active system, except for an additional delay while it waits to be polled. Also, people frequently log into active systems and request copying from one passive system to another. This requires two telephone calls, but even so, it is faster than mailing tapes.

Where convenient, we use hardwired communication lines. These permit much faster transmission and multiplexing of the communications link. Dial-up connections are made at either 300 or 1200 baud; hardwired connections are asynchronous up to 9600 baud and might run even faster on special-purpose communications hardware. fraser spider 1974 ieee fraser channel network

datamation 1975 Thus, systems typically join our network first as passive systems and when they find the service more important, they acquire automatic calling units and become active systems; eventually, they may install high-speed links to particular machines with which they handle a great deal of traffic. At no point, however, must users change their programs or procedures.

The basic operation of the network is very simple. Each participating system has a spool directory, in which work to be done (files to be moved, or commands to be executed remotely) is stored. A standard program, *uucico*, performs all transfers. This program starts by identifying a particular communication channel to a remote system with which it will hold a conversation. *Uucico* then selects a device and establishes the connection, logs onto the remote machine and starts the *uucico* program on the remote machine. Once two of these programs are connected, they first agree on a line protocol, and then start exchanging work. Each program in turn, beginning with the calling (active system) program, transmits everything it needs, and then asks the other what it wants done. Eventually neither has any more work, and both exit.

In this way, all services are available from all sites; passive sites, however, must wait until called. A variety of protocols may be used; this conforms to the real, non-standard world. As long as the caller and called programs have a protocol in common, they can communicate. Furthermore, each caller knows the hours when each destination system should be called. If a destination is unavailable, the data intended for it remain in the spool directory until the destination machine can be reached.

The implementation of this Bell Laboratories network between independent sites, all of which store proprietary programs and data, illustratives the pervasive need for security and administrative controls over file access. Each site, in configuring its programs and system files, limits and monitors transmission. In order to access a file a user needs access permission for the machine that contains the file and access permission for the file itself. This is achieved by first requiring the user to use his password to log into his local machine and then his local machine logs into the remote machine whose files are to be accessed. In addition, records are kept identifying all files that are moved into and out of the local system, and how the requestor of such accesses identified himself. Some sites may arrange to permit users only to call up and request work to be done; the calling users are then called back before the work is actually done. It is then possible to verify that the request is legitimate from the standpoint of the target system, as well as the originating system. Furthermore, because of the call-back, no site can masquerade as another even if it knows all the necessary passwords.

Each machine can optionally maintain a sequence count for conversations with other machines and require a verification of the count at the start of each conversation. Thus, even if call back is not in use, a successful masquerade requires the calling party to present the correct sequence number. A would-be impersonator must not just steal the correct phone number, user name, and password, but also the sequence count, and must call in sufficiently promptly to precede the next legitimate request from either side. Even a successful masquerade will be detected on the next correct conversation.

3. Processing

The user has two commands which set up communications, *uucp* to set up file copying, and *uux* to set up command execution where some of the required resources (system and/or files) are not on the local machine. Each of these commands will put work and data files into the spool directory for execution by *uucp* daemons. Figure 1 shows the major blocks of the file transfer process.

File Copy

The *uucico* program is used to perform all communications between the two systems. It performs the following functions:

- Scan the spool directory for work.

- Place a call to a remote system.
- Negotiate a line protocol to be used.
- Start program *uucico* on the remote system.
- Execute all requests from both systems.
- Log work requests and work completions.

Uucico may be started in several ways;

a) by a system daemon,

b) by one of the *uucp* or *uux* programs,

c) by a remote system.

Scan For Work

The file names in the spool directory are constructed to allow the daemon programs *(uucico, uuxqt)* to determine the files they should look at, the remote machines they should call and the order in which the files for a particular remote machine should be processed.

Call Remote System

The call is made using information from several files which reside in the uucp program directory. At the start of the call process, a lock is set on the system being called so that another call will not be attempted at the same time.

The system name is found in a "systems" file. The information contained for each system is:

[1] system name,

[2] times to call the system (days-of-week and times-of-day),

[3] device or device type to be used for call,

[4] line speed,

[5] phone number,

[6] login information (multiple fields).

The time field is checked against the present time to see if the call should be made. The *phone number* may contain abbreviations (e.g. "nyc", "boston") which get translated into dial sequences using a "dial-codes" file. This permits the same "phone number" to be stored at every site, despite local variations in telephone services and dialing conventions.

A "devices" file is scanned using fields [3] and [4] from the "systems" file to find an available device for the connection. The program will try all devices which satisfy [3] and [4] until a connection is made, or no more devices can be tried. If a non-multiplexable device is successfully opened, a lock file is created so that another copy of *uucico* will not try to use it. If the connection is complete, the *login information* is used to log into the remote system. Then a command is sent to the remote system to start the *uucico* program. The conversation between the two *uucico* programs begins with a handshake started by the called, *SLAVE*, system. The *SLAVE* sends a message to let the *MASTER* know it is ready to receive the system identification and conversation sequence number. The response from the *MASTER* is verified by the *SLAVE* and if acceptable, protocol selection begins.

Line Protocol Selection

The remote system sends a message

P*proto-list*

where *proto-list* is a string of characters, each representing a line protocol. The calling program checks the proto-list for a letter corresponding to an available line protocol and returns a *use-protocol* message. The *use-protocol* message is

Ucode

where code is either a one character protocol letter or a *N* which means there is no common protocol.

Greg Chesson designed and implemented the standard line protocol used by the uucp transmission program. Other protocols may be added by individual installations.

Work Processing

During processing, one program is the *MASTER* and the other is *SLAVE*. Initially, the calling program is the *MASTER*. These roles may switch one or more times during the conversation.

There are four messages used during the work processing, each specified by the first character of the message. They are

center; c l. S send a file, R receive a file, C copy complete, H hangup.

The *MASTER* will send *R* or *S* messages until all work from the spool directory is complete, at which point an *H* message will be sent. The *SLAVE* will reply with *SY*, *SN*, *RY*, *RN*, *HY*, *HN*, corresponding to *yes* or *no* for each request.

The send and receive replies are based on permission to access the requested file/directory. After each file is copied into the spool directory of the receiving system, a copy-complete message is sent by the receiver of the file. The message *CY* will be sent if the UNIX *cp* command, used to copy from the spool directory, is successful. Otherwise, a *CN* message is sent. The requests and results are logged on both systems, and, if requested, mail is sent to the user reporting completion (or the user can request status information from the log program at any time).

The hangup response is determined by the *SLAVE* program by a work scan of the spool directory. If work for the remote system exists in the *SLAVE's* spool directory, a *HN* message is sent and the programs switch roles. If no work exists, an *HY* response is sent.

A sample conversation is shown in Figure 2.

Conversation Termination

When a *HY* message is received by the *MASTER* it is echoed back to the *SLAVE* and the protocols are turned off. Each program sends a final "OO" message to the other.

4. Present Uses

One application of this software is remote mail. Normally, a UNIX system user writes "mail dan" to send mail to user "dan". By writing "mail usg!dan" the mail is sent to user "dan" on system "usg".

The primary uses of our network to date have been in software maintenance. Relatively few of the bytes passed between systems are intended for people to read. Instead, new programs (or new versions of programs) are sent to users, and potential bugs are returned to authors. Aaron Cohen has implemented a "stockroom" which allows remote users to call in and request software. He keeps a "stock list" of available programs, and new bug fixes and utilities are added regularly. In this way, users can always obtain the latest version of anything without bothering the authors of the programs. Although the stock list is maintained on a particular system, the items in the stockroom may be warehoused in many places; typically each program is distributed from the home site of its author. Where necessary, uucp does remote-to-remote copies.

We also routinely retrieve test cases from other systems to determine whether errors on remote systems are caused by local misconfigurations or old versions of software, or whether they are bugs that must be fixed at the home site. This helps identify errors rapidly. For one set of test programs maintained by us, over 70% of the bugs reported from remote sites were due to old software, and were fixed merely by distributing the current version.

Another application of the network for software maintenance is to compare files on two different machines. A very useful utility on one machine has been Doug McIlroy's "diff" program

which compares two text files and indicates the differences, line by line, between them. hunt mcilroy file Only lines which are not identical are printed. Similarly, the program "uudiff" compares files (or directories) on two machines. One of these directories may be on a passive system. The "uudiff" program is set up to work similarly to the inter-system mail, but it is slightly more complicated.

To avoid moving large numbers of usually identical files, *uudiff* computes file checksums on each side, and only moves files that are different for detailed comparison. For large files, this process can be iterated; checksums can be computed for each line, and only those lines that are different actually moved.

The "uux" command has been useful for providing remote output. There are some machines which do not have hard-copy devices, but which are connected over 9600 baud communication lines to machines with printers. The *uux* command allows the formatting of the printout on the local machine and printing on the remote machine using standard UNIX command programs.

5. Performance

Throughput, of course, is primarily dependent on transmission speed. The table below shows the real throughput of characters on communication links of different speeds. These numbers represent actual data transferred; they do not include bytes used by the line protocol for data validation such as checksums and messages. At the higher speeds, contention for the processors on both ends prevents the network from driving the line full speed. The range of speeds represents the difference between light and heavy loads on the two systems. If desired, operating system modifications can be installed that permit full use of even very fast links.

center; c c n n. Nominal speed Characters/sec. 300 baud 27 1200 baud 100-110 9600 baud 200-850

In addition to the transfer time, there is some overhead for making the connection and logging in ranging from 15 seconds to 1 minute. Even at 300 baud, however, a typical 5,000 byte source program can be transferred in four minutes instead of the 2 days that might be required to mail a tape.

Traffic between systems is variable. Between two closely related systems, we observed 20 files moved and 5 remote commands executed in a typical day. A more normal traffic out of a single system would be around a dozen files per day.

The total number of sites at present in the main network is 82, which includes most of the Bell Laboratories full-size machines which run the UNIX operating system. Geographically, the machines range from Andover, Massachusetts to Denver, Colorado.

Uucp has also been used to set up another network which connects a group of systems in operational sites with the home site. The two networks touch at one Bell Labs computer.

6. Further Goals

Eventually, we would like to develop a full system of remote software maintenance. Conventional maintenance (a support group which mails tapes) has many well-known disadvantages. brooks mythical man month 1975 There are distribution errors and delays, resulting in old software running at remote sites and old bugs continually reappearing. These difficulties are aggravated when there are 100 different small systems, instead of a few large ones.

The availability of file transfer on a network of compatible operating systems makes it possible just to send programs directly to the end user who wants them. This avoids the bottleneck of negotiation and packaging in the central support group. The "stockroom" serves this function for new utilities and fixes to old utilities. However, it is still likely that distributions will not be sent and installed as often as needed. Users are justifiably suspicious of the "latest version" that has just arrived; all too often it features the "latest bug." What is needed is to address both problems simultaneously:

1. Send distributions whenever programs change.
2. Have sufficient quality control so that users will install them.

To do this, we recommend systematic regression testing both on the distributing and receiving systems. Acceptance testing on the receiving systems can be automated and permits the local system to ensure that its essential work can continue despite the constant installation of changes sent from elsewhere. The work of writing the test sequences should be recovered in lower counseling and distribution costs.

Some slow-speed network services are also being implemented. We now have inter-system "mail" and "diff," plus the many implied commands represented by "uux." However, we still need inter-system "write" (real-time inter-user communication) and "who" (list of people logged in on different systems). A slow-speed network of this sort may be very useful for speeding up counseling and education, even if not fast enough for the distributed data base applications that attract many users to networks. Effective use of remote execution over slow-speed lines, however, must await the general installation of multiplexable channels so that long file transfers do not lock out short inquiries.

7. Lessons

The following is a summary of the lessons we learned in building these programs.

1. By starting your network in a way that requires no hardware or major operating system changes, you can get going quickly.
2. Support will follow use. Since the network existed and was being used, system maintainers were easily persuaded to help keep it operating, including purchasing additional hardware to speed traffic.
3. Make the network commands look like local commands. Our users have a resistance to learning anything new: all the inter-system commands look very similar to standard UNIX system commands so that little training cost is involved.
4. An initial error was not coordinating enough with existing communications projects: thus, the first version of this network was restricted to dial-up, since it did not support the various hardware links between systems. This has been fixed in the current system.

Acknowledgements

We thank G. L. Chesson for his design and implementation of the packet driver and protocol, and A. S. Cohen, J. Lions, and P. F. Long for their suggestions and assistance.

References

1. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 1905-1929, 1978.
2. G. L. Chesson, "The Network UNIX System," *Operating Systems Review*, vol. 9, no. 5, pp. 60-66, 1975. Also in *Proc. 5th Symp. on Operating Systems Principles.*
3. A. G. Fraser, "Spider — An Experimental Data Communications System," *Proc. IEEE Conf. on Communications*, p. 21F, June 1974. IEEE Cat. No. 74CH0859-9-CSCB.
4. A. G. Fraser, "A Virtual Channel Network," *Datamation*, pp. 51-56, February 1975.
5. J. W. Hunt and M. D. McIlroy, "An Algorithm for Differential File Comparison," Comp. Sci. Tech. Rep. No. 41, Bell Laboratories, Murray Hill, New Jersey, June 1976.
6. F. P. Brooks, Jr., *The Mythical Man-Month*, Addison-Wesley, Reading, Mass., 1975.

The C Programming Language - Reference Manual

Dennis M. Ritchie

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 execptions, 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	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$

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

$$\{\ \textit{expression}_{opt}\ \}$$

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^{wordsize}). 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 ($exp\text{-}1_{opt}$; $exp\text{-}2_{opt}$; $exp\text{-}3_{opt}$) *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

```
lw(2i) lw(2i) lw(2i).
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 **#defined** 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 [$constant\text{-}expression_{opt}$]

typedef-name:
identifier

16.3. Statements

compound-statement:
{ $declaration\text{-}list_{opt}$ $statement\text{-}list_{opt}$ }

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*"

Recent Changes to C

November 15, 1978

A few extensions have been made to the C language beyond what is described in the reference document ("The C Programming Language," Kerninghan and Ritchie, Prentice-Hall, 1978).

1. Structure assignment

Structures may be assigned, passed as arguments to functions, and returned by functions. The types of operands taking part must be the same. Other plausible operators, such as equality comparison, have not been implemented.

There is a subtle defect in the PDP-11 implementation of functions that return structures: if an interrupt occurs during the return sequence, and the same function is called reentrantly during the interrupt, the value returned form the first call may be corrupted. The problem can occur only in the presence of true interrupts, as in an operating system or a user program that makes significant use of signals; ordinary recursive calls are quite safe.

2. Enumeration type

There is a new data type analogous to the scalar types of Pascal. To the type-specifiers in the syntax on p. 193 of the C book add

enum-specifier

with syntax

enum-specifier:
enum (*enum-list*)
enum *identifier* (*enum-list*)
enum *identifier*

enum-list:
enumerator
enum-list , *enumerator*

enumerator:
identifier
identifier = *constant-expression*

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 enumerations. For example.

```
enum color ( chartreuse, burgundy, claret, winedark );
...
enum color *cp, col;
```

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 identifiers in the enum-list are declared as constants, and may appear wherever constants are required. If no enumerators with = appear, then the values of the 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.

Enumeration tags and constants must all be distinct, and, unlike structure tags and members, are drawn from the same set as ordinary identifiers.

Objects of given enumeration type are regarded as having a type distinct from objects of all other types, and *lint* flags type mismatches. In the PDP-11 implementation all enumeration variables are treated as if they were *int*.

Lint, a C Program Checker

S. C. Johnson

ABSTRACT

Lint is a command which examines C source programs, detecting a number of bugs and obscurities. It enforces the type rules of C more strictly than the C compilers. It may also be used to enforce a number of portability restrictions involved in moving programs between different machines and/or operating systems. Another option detects a number of wasteful, or error prone, constructions which nevertheless are, strictly speaking, legal.

Lint accepts multiple input files and library specifications, and checks them for consistency.

The separation of function between *lint* and the C compilers has both historical and practical rationale. The compilers turn C programs into executable files rapidly and efficiently. This is possible in part because the compilers do not do sophisticated type checking, especially between separately compiled programs. *Lint* takes a more global, leisurely view of the program, looking much more carefully at the compatibilities.

This document discusses the use of *lint*, gives an overview of the implementation, and gives some hints on the writing of machine independent C code.

July 26, 1978

Lint, a C Program Checker

S. C. Johnson

Introduction and Usage

Suppose there are two C[1] source files, *file1.*c and *file2.c*, which are ordinarily compiled and loaded together. Then the command

```
lint file1.c file2.c
```

produces messages describing inconsistencies and inefficiencies in the programs. The program enforces the typing rules of C more strictly than the C compilers (for both historical and practical reasons) enforce them. The command

```
lint -p file1.c file2.c
```

will produce, in addition to the above messages, additional messages which relate to the portability of the programs to other operating systems and machines. Replacing the **-p** by **-h** will produce messages about various error-prone or wasteful constructions which, strictly speaking, are not bugs. Saying **-hp** gets the whole works.

The next several sections describe the major messages; the document closes with sections discussing the implementation and giving suggestions for writing portable C. An appendix gives a summary of the *lint* options.

A Word About Philosophy

Many of the facts which *lint* needs may be impossible to discover. For example, whether a given function in a program ever gets called may depend on the input data. Deciding whether *exit* is ever called is equivalent to solving the famous "halting problem," known to be recursively undecidable.

Thus, most of the *lint* algorithms are a compromise. If a function is never mentioned, it can never be called. If a function is mentioned, *lint* assumes it can be called; this is not necessarily so, but in practice is quite reasonable.

Lint tries to give information with a high degree of relevance. Messages of the form "*xxx* might be a bug" are easy to generate, but are acceptable only in proportion to the fraction of real bugs they uncover. If this fraction of real bugs is too small, the messages lose their credibility and serve merely to clutter up the output, obscuring the more important messages.

Keeping these issues in mind, we now consider in more detail the classes of messages which *lint* produces.

Unused Variables and Functions

As sets of programs evolve and develop, previously used variables and arguments to functions may become unused; it is not uncommon for external variables, or even entire functions, to become unnecessary, and yet not be removed from the source. These "errors of commission" rarely cause working programs to fail, but they are a source of inefficiency, and make programs harder to understand and change. Moreover, information about such unused variables and functions can occasionally serve to discover bugs; if a function does a necessary job, and is never called, something is wrong!

Lint complains about variables and functions which are defined but not otherwise mentioned. An exception is variables which are declared through explicit **extern** statements but are never

referenced; thus the statement

```
extern float sin( );
```

will evoke no comment if *sin* is never used. Note that this agrees with the semantics of the C compiler. In some cases, these unused external declarations might be of some interest; they can be discovered by adding the **-x** flag to the *lint* invocation.

Certain styles of programming require many functions to be written with similar interfaces; frequently, some of the arguments may be unused in many of the calls. The **-v** option is available to suppress the printing of complaints about unused arguments. When **-v** is in effect, no messages are produced about unused arguments except for those arguments which are unused and also declared as register arguments; this can be considered an active (and preventable) waste of the register resources of the machine.

There is one case where information about unused, or undefined, variables is more distracting than helpful. This is when *lint* is applied to some, but not all, files out of a collection which are to be loaded together. In this case, many of the functions and variables defined may not be used, and, conversely, many functions and variables defined elsewhere may be used. The **-u** flag may be used to suppress the spurious messages which might otherwise appear.

Set/Used Information

Lint attempts to detect cases where a variable is used before it is set. This is very difficult to do well; many algorithms take a good deal of time and space, and still produce messages about perfectly valid programs. *Lint* detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a "use," since the actual use may occur at any later time, in a data dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement, since the true flow of control need not be discovered. It does mean that *lint* can complain about some programs which are legal, but these programs would probably be considered bad on stylistic grounds (e.g. might contain at least two **goto**'s). Because static and external variables are initialized to 0, no meaningful information can be discovered about their uses. The algorithm deals correctly, however, with initialized automatic variables, and variables which are used in the expression which first sets them.

The set/used information also permits recognition of those local variables which are set and never used; these form a frequent source of inefficiencies, and may also be symptomatic of bugs.

Flow of Control

Lint attempts to detect unreachable portions of the programs which it processes. It will complain about unlabeled statements immediately following **goto**, **break**, **continue**, or **return** statements. An attempt is made to detect loops which can never be left at the bottom, detecting the special cases **while**(1) and **for**(;;) as infinite loops. *Lint* also complains about loops which cannot be entered at the top; some valid programs may have such loops, but at best they are bad style, at worst bugs.

Lint has an important area of blindness in the flow of control algorithm: it has no way of detecting functions which are called and never return. Thus, a call to *exit* may cause unreachable code which *lint* does not detect; the most serious effects of this are in the determination of returned function values (see the next section).

One form of unreachable statement is not usually complained about by *lint;* a **break** statement that cannot be reached causes no message. Programs generated by *yacc*,[2] and especially *lex*,[3] may have literally hundreds of unreachable **break** statements. The **-O** flag in the C compiler will often eliminate the resulting object code inefficiency. Thus, these unreached statements are of little importance, there is typically nothing the user can do about them, and the resulting messages would clutter up the *lint* output. If these messages are desired, *lint* can be invoked with

the **–b** option.

Function Values

Sometimes functions return values which are never used; sometimes programs incorrectly use function "values" which have never been returned. *Lint* addresses this problem in a number of ways.

Locally, within a function definition, the appearance of both

```
return( expr );
```

and

```
return ;
```

statements is cause for alarm; *lint* will give the message

function *name* contains return(e) and return

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

```
f ( a ) {
        if ( a ) return ( 3 );
        g ( );
        }
```

Notice that, if *a* tests false, *f* will call *g* and then return with no defined return value; this will trigger a complaint from *lint*. If *g*, like *exit*, never returns, the message will still be produced when in fact nothing is wrong.

In practice, some potentially serious bugs have been discovered by this feature; it also accounts for a substantial fraction of the "noise" messages produced by *lint*.

On a global scale, *lint* detects cases where a function returns a value, but this value is sometimes, or always, unused. When the value is always unused, it may constitute an inefficiency in the function definition. When the value is sometimes unused, it may represent bad style (e.g., not testing for error conditions).

The dual problem, using a function value when the function does not return one, is also detected. This is a serious problem. Amazingly, this bug has been observed on a couple of occasions in "working" programs; the desired function value just happened to have been computed in the function return register!

Type Checking

Lint enforces the type checking rules of C more strictly than the compilers do. The additional checking is in four major areas: across certain binary operators and implied assignments, at the structure selection operators, between the definition and uses of functions, and in the use of enumerations.

There are a number of operators which have an implied balancing between types of the operands. The assignment, conditional (? :), and relational operators have this property; the argument of a **return** statement, and expressions used in initialization also suffer similar conversions. In these operations, **char**, **short**, **int**, **long**, **unsigned**, **float**, and **double** types may be freely intermixed. The types of pointers must agree exactly, except that arrays of *x*'s can, of course, be intermixed with pointers to *x*'s.

The type checking rules also require that, in structure references, the left operand of the —> be a pointer to structure, the left operand of the . be a structure, and the right operand of these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types **float** and **double** may be freely matched, as may the types **char**, **short**, **int**, and **unsigned**. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts.

With enumerations, checks are made that enumeration variables or members are not mixed with other types, or other enumerations, and that the only operations applied are =, initialization, ==, !=, and function arguments and return values.

Type Casts

The type cast feature in C was introduced largely as an aid to producing more portable programs. Consider the assignment

```
p = 1 ;
```

where *p* is a character pointer. *Lint* will quite rightly complain. Now, consider the assignment

```
p = (char *)1 ;
```

in which a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong motivation for doing this, and has clearly signaled his intentions. It seems harsh for *lint* to continue to complain about this. On the other hand, if this code is moved to another machine, such code should be looked at carefully. The **-c** flag controls the printing of comments about casts. When **-c** is in effect, casts are treated as though they were assignments subject to complaint; otherwise, all legal casts are passed without comment, no matter how strange the type mixing seems to be.

Nonportable Character Use

On the PDP-11, characters are signed quantities, with a range from -128 to 127. On most of the other C implementations, characters take on only positive values. Thus, *lint* will flag certain comparisons and assignments as being illegal or nonportable. For example, the fragment

```
char c;
    ...
if( (c = getchar( )) < 0 ) ....
```

works on the PDP-11, but will fail on machines where characters always take on positive values. The real solution is to declare *c* an integer, since *getchar* is actually returning integer values. In any case, *lint* will say "nonportable character comparison".

A similar issue arises with bitfields; when assignments of constant values are made to bitfields, the field may be too small to hold the value. This is especially true because on some machines bitfields are considered as signed quantities. While it may seem unintuitive to consider that a two bit field declared of type **int** cannot hold the value 3, the problem disappears if the bitfield is declared to have type **unsigned**.

Assignments of longs to ints

Bugs may arise from the assignment of **long** to an **int**, which loses accuracy. This may happen in programs which have been incompletely converted to use **typedefs**. When a **typedef** variable is changed from **int** to **long**, the program can stop working because some intermediate results may be assigned to **ints**, losing accuracy. Since there are a number of legitimate reasons for assigning **longs** to **ints**, the detection of these assignments is enabled by the **-a** flag.

Strange Constructions

Several perfectly legal, but somewhat strange, constructions are flagged by *lint;* the messages hopefully encourage better code quality, clearer style, and may even point out bugs. The **-h** flag is used to enable these checks. For example, in the statement

```
*p++ ;
```

the * does nothing; this provokes the message "null effect" from *lint*. The program fragment

```
unsigned x ;
if( x < 0 ) ...
```

is clearly somewhat strange; the test will never succeed. Similarly, the test

```
if( x > 0 ) ...
```

is equivalent to

```
if( x != 0 )
```

which may not be the intended action. *Lint* will say "degenerate unsigned comparison" in these cases. If one says

```
if( 1 != 0 ) ....
```

lint will report "constant in conditional context", since the comparison of 1 with 0 gives a constant result.

Another construction detected by *lint* involves operator precedence. Bugs which arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statements

```
if( x&077 == 0 ) ...
```

or

```
x<<2 + 40
```

probably do not do what was intended. The best solution is to parenthesize such expressions, and *lint* encourages this by an appropriate message.

Finally, when the **-h** flag is in force *lint* complains about variables which are redeclared in inner blocks in a way that conflicts with their use in outer blocks. This is legal, but is considered by many (including the author) to be bad style, usually unnecessary, and frequently a bug.

Ancient History

There are several forms of older syntax which are being officially discouraged. These fall into two classes, assignment operators and initialization.

The older forms of assignment operators (e.g., =+, =-, . . .) could cause ambiguous expressions, such as

```
a =-1 ;
```

which could be taken as either

```
a =- 1 ;
```

or

```
a = -1 ;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer, and preferred operators (+=, -=, etc.) have no such ambiguities. To spur the abandonment of the older forms, *lint* complains about these old fashioned operators.

A similar issue arises with initialization. The older language allowed

```
int x 1 ;
```

to initialize *x* to 1. This also caused syntactic difficulties: for example,

```
int x (-1);
```

looks somewhat like the beginning of a function declaration:

```
int x (y){ ...
```

and the compiler must read a fair ways past *x* in order to sure what the declaration really is.. Again, the problem is even more perplexing when the initializer involves a macro. The current syntax places an equals sign between the variable and the initializer:

```
int x = -1;
```

This is free of any possible syntactic ambiguity.

Pointer Alignment

Certain pointer assignments may be reasonable on some machines, and illegal on others, due entirely to alignment restrictions. For example, on the PDP-11, it is reasonable to assign integer pointers to double pointers, since double precision values may begin on any integer boundary. On the Honeywell 6000, double precision values must begin on even word boundaries; thus, not all such assignments make sense. *Lint* tries to detect cases where pointers are assigned to other pointers, and such alignment problems might arise. The message "possible pointer alignment problem" results from this situation whenever either the **-p** or **-h** flags are in effect.

Multiple Uses and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine dependent. For example, on machines (like the PDP-11) in which the stack runs backwards, function arguments will probably be best evaluated from right-to-left; on machines with a stack running forward, left-to-right seems most attractive. Function calls embedded as arguments of other functions may or may not be treated similarly to ordinary arguments. Similar issues arise with other operators which have side effects, such as the assignment operators and the increment and decrement operators.

In order that the efficiency of C on a particular machine not be unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler, and, in fact, the various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect, and also used elsewhere in the same expression, the result is explicitly undefined.

Lint checks for the important special case where a simple scalar variable is affected. For example, the statement

```
a[i] = b[i++];
```

will draw the complaint:

```
warning: i evaluation order undefined
```

Implementation

Lint consists of two programs and a driver. The first program is a version of the Portable C Compiler[4,5] which is the basis of the IBM 370, Honeywell 6000, and Interdata 8/32 C compilers. This compiler does lexical and syntax analysis on the input text, constructs and maintains symbol tables, and builds trees for expressions. Instead of writing an intermediate file which is passed to a code generator, as the other compilers do, *lint* produces an intermediate file which consists of lines of ascii text. Each line contains an external variable name, an encoding of the context in which it was seen (use, definition, declaration, etc.), a type specifier, and a source file name and line number. The information about variables local to a function or file is collected by accessing the symbol table, and examining the expression trees.

Comments about local problems are produced as detected. The information about external names is collected onto an intermediate file. After all the source files and library descriptions have been collected, the intermediate file is sorted to bring all information collected about a given external name together. The second, rather small, program then reads the lines from the intermediate file and compares all of the definitions, declarations, and uses for consistency.

The driver controls this process, and is also responsible for making the options available to both passes of *lint*.

Portability

C on the Honeywell and IBM systems is used, in part, to write system code for the host operating system. This means that the implementation of C tends to follow local conventions rather than adhere strictly to UNIX† system conventions. Despite these differences, many C programs have been successfully moved to GCOS and the various IBM installations with little effort. This section describes some of the differences between the implementations, and discusses the *lint* features which encourage portability.

Uninitialized external variables are treated differently in different implementations of C. Suppose two files both contain a declaration without initialization, such as

```
int a ;
```

outside of any function. The UNIX loader will resolve these declarations, and cause only a single word of storage to be set aside for *a*. Under the GCOS and IBM implementations, this is not feasible (for various stupid reasons!) so each such declaration causes a word of storage to be set aside and called *a*. When loading or library editing takes place, this causes fatal conflicts which prevent the proper operation of the program. If *lint* is invoked with the **-p** flag, it will detect such multiple definitions.

A related difficulty comes from the amount of information retained about external names during the loading process. On the UNIX system, externally known names have seven significant characters, with the upper/lower case distinction kept. On the IBM systems, there are eight significant characters, but the case distinction is lost. On GCOS, there are only six characters, of a single case. This leads to situations where programs run on the UNIX system, but encounter loader problems on the IBM or GCOS systems. *Lint* **-p** causes all external symbols to be mapped to one case and truncated to six characters, providing a worst-case analysis.

A number of differences arise in the area of character handling: characters in the UNIX system are eight bit ascii, while they are eight bit ebcdic on the IBM, and nine bit ascii on GCOS. Moreover, character strings go from high to low bit positions ("left to right") on GCOS and IBM, and low to high ("right to left") on the PDP-11. This means that code attempting to construct strings out of character constants, or attempting to use characters as indices into arrays, must be looked at with great suspicion. *Lint* is of little help here, except to flag multi-character character constants.

Of course, the word sizes are different! This causes less trouble than might be expected, at least when moving from the UNIX system (16 bit words) to the IBM (32 bits) or GCOS (36 bits). The main problems are likely to arise in shifting or masking. C now supports a bit-field facility, which can be used to write much of this code in a reasonably portable way. Frequently, portability of such code can be enhanced by slight rearrangements in coding style. Many of the incompatibilities seem to have the flavor of writing

```
x &= 0177700 ;
```

to clear the low order six bits of *x*. This suffices on the PDP-11, but fails badly on GCOS and IBM. If the bit field feature cannot be used, the same effect can be obtained by writing

† UNIX is a trademark of Bell Laboratories.

```
x &= ~ 077 ;
```

which will work on all these machines.

The right shift operator is arithmetic shift on the PDP-11, and logical shift on most other machines. To obtain a logical shift on all machines, the left operand can be typed **unsigned**. Characters are considered signed integers on the PDP-11, and unsigned on the other machines. This persistence of the sign bit may be reasonably considered a bug in the PDP-11 hardware which has infiltrated itself into the C language. If there were a good way to discover the programs which would be affected, C could be changed; in any case, *lint* is no help here.

The above discussion may have made the problem of portability seem bigger than it in fact is. The issues involved here are rarely subtle or mysterious, at least to the implementor of the program, although they can involve some work to straighten out. The most serious bar to the portability of UNIX system utilities has been the inability to mimic essential UNIX system functions on the other systems. The inability to seek to a random character position in a text file, or to establish a pipe between processes, has involved far more rewriting and debugging than any of the differences in C compilers. On the other hand, *lint* has been very helpful in moving the UNIX operating system and associated utility programs to other machines.

Shutting Lint Up

There are occasions when the programmer is smarter than *lint*. There may be valid reasons for "illegal" type casts, functions with a variable number of arguments, etc. Moreover, as specified above, the flow of control information produced by *lint* often has blind spots, causing occasional spurious messages about perfectly reasonable programs. Thus, some way of communicating with *lint*, typically to shut it up, is desirable.

The form which this mechanism should take is not at all clear. New keywords would require current and old compilers to recognize these keywords, if only to ignore them. This has both philosophical and practical problems. New preprocessor syntax suffers from similar problems.

What was finally done was to cause a number of words to be recognized by *lint* when they were embedded in comments. This required minimal preprocessor changes; the preprocessor just had to agree to pass comments through to its output, instead of deleting them as had been previously done. Thus, *lint* directives are invisible to the compilers, and the effect on systems with the older preprocessors is merely that the *lint* directives don't work.

The first directive is concerned with flow of control information; if a particular place in the program cannot be reached, but this is not apparent to *lint*, this can be asserted by the directive

```
/* NOTREACHED */
```

at the appropriate spot in the program. Similarly, if it is desired to turn off strict type checking for the next expression, the directive

```
/* NOSTRICT */
```

can be used; the situation reverts to the previous default after the next expression. The **-v** flag can be turned on for one function by the directive

```
/* ARGSUSED */
```

Complaints about variable number of arguments in calls to a function can be turned off by the directive

```
/* VARARGS */
```

preceding the function definition. In some cases, it is desirable to check the first several arguments, and leave the later arguments unchecked. This can be done by following the VARARGS keyword immediately with a digit giving the number of arguments which should be checked; thus,

/* VARARGS2 */

will cause the first two arguments to be checked, the others unchecked. Finally, the directive

/* LINTLIBRARY */

at the head of a file identifies this file as a library declaration file; this topic is worth a section by itself.

Library Declaration Files

Lint accepts certain library directives, such as

-ly

and tests the source files for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library directives. These files all begin with the directive

/* LINTLIBRARY */

which is followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function return type, whether the dummy function returns a value, and the number and types of arguments to the function. The VARARGS and ARGSUSED directives can be used to specify features of the library functions.

Lint library files are processed almost exactly like ordinary source files. The only difference is that functions which are defined on a library file, but are not used on a source file, draw no complaints. *Lint* does not simulate a full library search algorithm, and complains if the source files contain a redefinition of a library routine (this is a feature!).

By default, *lint* checks the programs it is given against a standard library file, which contains descriptions of the programs which are normally loaded when a C program is run. When the **-p** flag is in effect, another file is checked containing descriptions of the standard I/O library routines which are expected to be portable across various machines. The **-n** flag can be used to suppress all library checking.

Bugs, etc.

Lint was a difficult program to write, partially because it is closely connected with matters of programming style, and partially because users usually don't notice bugs which cause *lint* to miss errors which it should have caught. (By contrast, if *lint* incorrectly complains about something that is correct, the programmer reports that immediately!)

A number of areas remain to be further developed. The checking of structures and arrays is rather inadequate; size incompatibilities go unchecked, and no attempt is made to match up structure and union declarations across files. Some stricter checking of the use of the **typedef** is clearly desirable, but what checking is appropriate, and how to carry it out, is still to be determined.

Lint shares the preprocessor with the C compiler. At some point it may be appropriate for a special version of the preprocessor to be constructed which checks for things such as unused macro definitions, macro arguments which have side effects which are not expanded at all, or are expanded more than once, etc.

The central problem with *lint* is the packaging of the information which it collects. There are many options which serve only to turn off, or slightly modify, certain features. There are pressures to add even more of these options.

In conclusion, it appears that the general notion of having two programs is a good one. The compiler concentrates on quickly and accurately turning the program text into bits which can be run; *lint* concentrates on issues of portability, style, and efficiency. *Lint* can afford to be wrong, since incorrectness and over-conservatism are merely annoying, not fatal. The compiler can be fast since it knows that *lint* will cover its flanks. Finally, the programmer can concentrate at one stage

of the programming process solely on the algorithms, data structures, and correctness of the program, and then later retrofit, with the aid of *lint*, the desirable properties of universality and portability.

References

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
2. S. C. Johnson, "Yacc — Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey, July 1975.
3. M. E. Lesk, "Lex — A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey, October 1975.
4. S. C. Johnson and D. M. Ritchie, "UNIX Time-Sharing System: Portability of C Programs and the UNIX System," *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 2021-2048, 1978.
5. S. C. Johnson, "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages,* pp. 97-104, January 1978.

Appendix: Current Lint Options

The command currently has the form

lint [–options] files... library-descriptors...

The options are

h Perform heuristic checks

p Perform portability checks

v Don't report unused arguments

u Don't report unused or undefined externals

b Report unreachable **break** statements.

x Report unused external declarations

a Report assignments of **long** to **int** or shorter.

c Complain about questionable casts

n No library checking is done

s Same as **h** (for historical reasons)

A Tour through the UNIX† C Compiler

D. M. Ritchie

Languages
Computing

The Intermediate Language

Communication between the two phases of the compiler proper is carried out by means of a pair of intermediate files. These files are treated as having identical structure, although the second file contains only the code generated for strings. It is convenient to write strings out separately to reduce the need for multiple location counters in a later assembly phase.

The intermediate language is not machine-independent; its structure in a number of ways reflects the fact that C was originally a one-pass compiler chopped in two to reduce the maximum memory requirement. In fact, only the latest version of the compiler has a complete intermediate language at all. Until recently, the first phase of the compiler generated assembly code for those constructions it could deal with, and passed expression parse trees, in absolute binary form, to the second phase for code generation. Now, at least, all inter-phase information is passed in a describable form, and there are no absolute pointers involved, so the coupling between the phases is not so strong.

The areas in which the machine (and system) dependencies are most noticeable are

1. Storage allocation for automatic variables and arguments has already been performed, and nodes for such variables refer to them by offset from a display pointer. Type conversion (for example, from integer to pointer) has already occurred using the assumption of byte addressing and 2-byte words.
2. Data representations suitable to the PDP-11 are assumed; in particular, floating point constants are passed as four words in the machine representation.

As it happens, each intermediate file is represented as a sequence of binary numbers without any explicit demarcations. It consists of a sequence of conceptual lines, each headed by an operator, and possibly containing various operands. The operators are small numbers; to assist in recognizing failure in synchronization, the high-order byte of each operator word is always the octal number 376. Operands are either 16-bit binary numbers or strings of characters representing names. Each name is terminated by a null character. There is no alignment requirement for numerical operands and so there is no padding after a name string.

The binary representation was chosen to avoid the necessity of converting to and from character form and to minimize the size of the files. It would be very easy to make each operator-operand 'line' in the file be a genuine, printable line, with the numbers in octal or decimal; this in fact was the representation originally used.

The operators fall naturally into two classes: those which represent part of an expression, and all others. Expressions are transmitted in a reverse-Polish notation; as they are being read, a tree is built which is isomorphic to the tree constructed in the first phase. Expressions are passed as a whole, with no non-expression operators intervening. The reader maintains a stack; each leaf of the expression tree (name, constant) is pushed on the stack; each unary operator replaces the top of the stack by a node whose operand is the old top-of-stack; each binary operator replaces the top

pair on the stack with a single entry. When the expression is complete there is exactly one item on the stack. Following each expression is a special operator which passes the unique previous expression to the 'optimizer' described below and then to the code generator.

Here is the list of operators not themselves part of expressions.

EOF

marks the end of an input file.

BDATA *flag data ...*

specifies a sequence of bytes to be assembled as static data. It is followed by pairs of words; the first member of the pair is non-zero to indicate that the data continue; a zero flag is not followed by data and terminates the operator. The data bytes occupy the low-order part of a word.

WDATA *flag data ...*

specifies a sequence of words to be assembled as static data; it is identical to the BDATA operator except that entire words, not just bytes, are passed.

PROG

means that subsequent information is to be compiled as program text.

DATA

means that subsequent information is to be compiled as static data.

BSS

means that subsequent information is to be compiled as unitialized static data.

SYMDEF *name*

means that the symbol *name* is an external name defined in the current program. It is produced for each external data or function definition.

CSPACE *name size*

indicates that the name refers to a data area whose size is the specified number of bytes. It is produced for external data definitions without explicit initialization.

SSPACE *size*

indicates that *size* bytes should be set aside for data storage. It is used to pad out short initializations of external data and to reserve space for static (internal) data. It will be preceded by an appropriate label.

EVEN

is produced after each external data definition whose size is not an integral number of words. It is not produced after strings except when they initialize a character array.

NLABEL *name*

is produced just before a BDATA or WDATA initializing external data, and serves as a label for the data.

RLABEL *name*

is produced just before each function definition, and labels its entry point.

SNAME *name number*

is produced at the start of each function for each static variable or label declared therein. Subsequent uses of the variable will be in terms of the given number. The code generator uses this only to produce a debugging symbol table.

ANAME *name number*

Likewise, each automatic variable's name and stack offset is specified by this operator. Arguments count as automatics.

RNAME *name number*

Each register variable is similarly named, with its register number.

SAVE *number*

produces a register-save sequence at the start of each function, just after its label (RLABEL).

SETREG *number*

is used to indicate the number of registers used for register variables. It actually gives the register number of the lowest free register; it is redundant because the RNAME operators could be counted instead.

PROFIL

is produced before the save sequence for functions when the profile option is turned on. It produces code to count the number of times the function is called.

SWIT *deflab line label value ...*

is produced for switches. When control flows into it, the value being switched on is in the register forced by RFORCE (below). The switch statement occurred on the indicated line of the source, and the label number of the default location is *deflab*. Then the operator is followed by a sequence of label-number and value pairs; the list is terminated by a 0 label.

LABEL *number*

generates an internal label. It is referred to elsewhere using the given number.

BRANCH *number*

indicates an unconditional transfer to the internal label number given.

RETRN

produces the return sequence for a function. It occurs only once, at the end of each function.

EXPR *line*

causes the expression just preceding to be compiled. The argument is the line number in the source where the expression occurred.

NAME *class type name*

NAME *class type number*

indicates a name occurring in an expression. The first form is used when the name is external; the second when the name is automatic, static, or a register. Then the number indicates the stack offset, the label number, or the register number as appropriate. Class and type encoding is described elsewhere.

CON *type value*

transmits an integer constant. This and the next two operators occur as part of expressions.

FCON *type 4-word-value*

transmits a floating constant as four words in PDP-11 notation.

SFCON *type value*

transmits a floating-point constant whose value is correctly represented by its high-order word in PDP-11 notation.

NULL

indicates a null argument list of a function call in an expression; call is a binary operator whose second operand is the argument list.

CBRANCH *label cond*

produces a conditional branch. It is an expression operator, and will be followed by an EXPR. The branch to the label number takes place if the expression's truth value is the same as that of *cond*. That is, if *cond=1* and the expression evaluates to true, the branch is taken.

binary-operator *type*

There are binary operators corresponding to each such source-language operator; the type of the result of each is passed as well. Some perhaps-unexpected ones are: COMMA, which is a right-associative operator designed to simplify right-to-left evaluation of function arguments; prefix and postfix ++ and —, whose second operand is the increment amount, as a CON; QUEST and COLON, to express the conditional expression as 'a?(b:c)'; and a sequence of special operators for expressing relations between pointers, in case pointer comparison is different from integer comparison (e.g. unsigned).

unary-operator *type*

There are also numerous unary operators. These include ITOF, FTOI, FTOL, LTOF, ITOL, LTOI which convert among floating, long, and integer; JUMP which branches indirectly through a label expression; INIT, which compiles the value of a constant expression used as an initializer; RFORCE, which is used before a return sequence or a switch to place a value in an agreed-upon register.

Expression Optimization

Each expression tree, as it is read in, is subjected to a fairly comprehensive analysis. This is performed by the *optim* routine and a number of subroutines; the major things done are

1. Modifications and simplifications of the tree so its value may be computed more efficiently and conveniently by the code generator.
2. Marking each interior node with an estimate of the number of registers required to evaluate it. This register count is needed to guide the code generation algorithm.

One thing that is definitely not done is discovery or exploitation of common subexpressions, nor is this done anywhere in the compiler.

The basic organization is simple: a depth-first scan of the tree. *Optim* does nothing for leaf nodes (except for automatics; see below), and calls ***unoptim*** to handle unary operators. For binary operators, it calls itself to process the operands, then treats each operator separately. One important case is commutative and associative operators, which are handled by ***acommute***.

Here is a brief catalog of the transformations carried out by by *optim* itself. It is not intended to be complete. Some of the transformations are machine-dependent, although they may well be useful on machines other than the PDP-11.

1. As indicated in the discussion of ***unoptim*** below, the optimizer can create a node type corresponding to the location addressed by a register plus a constant offset. Since this is precisely the implementation of automatic variables and arguments, where the register is fixed by convention, such variables are changed to the new form to simplify later processing.
2. Associative and commutative operators are processed by the special routine ***acommute***.
3. After processing by ***acommute***, the bitwise & operator is turned into a new ***andn*** operator; 'a & b' becomes 'a ***andn*** ~b'. This is done because the PDP-11 provides no ***and*** operator, but only ***andn***. A similar transformation takes place for '=&'.
4. Relationals are turned around so the more complicated expression is on the left. (So that '2 > f(x)' becomes 'f(x) < 2'). This improves code generation since the algorithm prefers to have the right operand require fewer registers than the left.
5. An expression minus a constant is turned into the expression plus the negative constant, and the ***acommute*** routine is called to take advantage of the properties of addition.
6. Operators with constant operands are evaluated.
7. Right shifts (unless by 1) are turned into left shifts with a negated right operand, since the PDP-11 lacks a general right-shift operator.
8. A number of special cases are simplified, such as division or multiplication by 1, and shifts by 0.

The ***unoptim*** routine performs the same sort of processing for unary operators.

1. '*&x' and '&*x' are simplified to 'x'.
2. If *r* is a register and *c* is a constant or the address of a static or external variable, the expressions '*(r+c)' and '*r' are turned into a special kind of name node which expresses the name itself and the offset. This simplifies subsequent processing because such constructions can appear as the the address of a PDP-11 instruction.
3. When the unary '&' operator is applied to a name node of the special kind just discussed, it is reworked to make the addition explicit again; this is done because the PDP-11 has no 'load address' instruction.
4. Constructions like '*r++' and '*—r' where *r* is a register are discovered and marked as being implementable using the PDP-11 auto-increment and -decrement modes.
5. If '!' is applied to a relational, the '!' is discarded and the sense of the relational is reversed.
6. Special cases involving reflexive use of negation and complementation are discovered.
7. Operations applying to constants are evaluated.

The ***acommute*** routine, called for associative and commutative operators, discovers clusters of the same operator at the top levels of the current tree, and arranges them in a list: for 'a+((b+c)+(d+f))' the list would be'a,b,c,d,e,f'. After each subtree is optimized, the list is sorted in decreasing difficulty of computation; as mentioned above, the code generation algorithm works best when left operands are the difficult ones. The 'degree of difficulty' computed is actually finer than the mere number of registers required; a constant is considered simpler than the address of a static or external, which is simpler than reference to a variable. This makes it easy to fold all the constants together, and also to merge together the sum of a constant and the address of a static or external (since in such nodes there is space for an 'offset' value). There are also special cases, like multiplication by 1 and addition of 0.

A special routine is invoked to handle sums of products. *Distrib* is based on the fact that it is better to compute 'c1*c2*x + c1*y' as 'c1*(c2*x + y)' and makes the divisibility tests required to assure the correctness of the transformation. This transformation is rarely possible with code directly written by the user, but it invariably occurs as a result of the implementation of multi-dimensional arrays.

Finally, *acommute* reconstructs a tree from the list of expressions which result.

Code Generation

The grand plan for code-generation is independent of any particular machine; it depends largely on a set of tables. But this fact does not necessarily make it very easy to modify the compiler to produce code for other machines, both because there is a good deal of machine-dependent structure in the tables, and because in any event such tables are non-trivial to prepare.

The arguments to the basic code generation routine *rcexpr* are a pointer to a tree representing an expression, the name of a code-generation table, and the number of a register in which the value of the expression should be placed. *Rcexpr* returns the number of the register in which the value actually ended up; its caller may need to produce a *mov* instruction if the value really needs to be in the given register. There are four code generation tables.

Regtab is the basic one, which actually does the job described above: namely, compile code which places the value represented by the expression tree in a register.

Cctab is used when the value of the expression is not actually needed, but instead the value of the condition codes resulting from evaluation of the expression. This table is used, for example, to evaluate the expression after *if*. It is clearly silly to calculate the value (0 or 1) of the expression 'a==b' in the context 'if (a==b) ... '

The *sptab* table is used when the value of an expression is to be pushed on the stack, for example when it is an actual argument. For example in the function call 'f(a)' it is a bad idea to load *a* into a register which is then pushed on the stack, when there is a single instruction which does the job.

The *efftab* table is used when an expression is to be evaluated for its side effects, not its value. This occurs mostly for expressions which are statements, which have no value. Thus the code for the statement 'a = b' need produce only the approoriate *mov* instruction, and need not leave the value of *b* in a register, while in the expression 'a + (b = c)' the value of 'b = c' will appear in a register.

All of the tables besides *regtab* are rather small, and handle only a relatively few special cases. If one of these subsidiary tables does not contain an entry applicable to the given expression tree, *rcexpr* uses *regtab* to put the value of the expression into a register and then fixes things up; nothing need be done when the table was *efftab*, but a *tst* instruction is produced when the table called for was *cctab*, and a *mov* instruction, pushing the register on the stack, when the table was *sptab*.

The *rcexpr* routine itself picks off some special cases, then calls *cexpr* to do the real work. *Cexpr* tries to find an entry applicable to the given tree in the given table, and returns –1 if no such entry is found, letting *rcexpr* try again with a different table. A successful match yields a string containing both literal characters which are written out and pseudo-operations, or macros, which are expanded. Before studying the contents of these strings we will consider how table entries are matched against trees.

Recall that most non-leaf nodes in an expression tree contain the name of the operator, the type of the value represented, and pointers to the subtrees (operands). They also contain an estimate of the number of registers required to evaluate the expression, placed there by the expression-optimizer routines. The register counts are used to guide the code generation process, which is based on the Sethi-Ullman algorithm.

The main code generation tables consist of entries each containing an operator number and a pointer to a subtable for the corresponding operator. A subtable consists of a sequence of entries,

each with a key describing certain properties of the operands of the operator involved; associated with the key is a code string. Once the subtable corresponding to the operator is found, the subtable is searched linearly until a key is found such that the properties demanded by the key are compatible with the operands of the tree node. A successful match returns the code string; an unsuccessful search, either for the operator in the main table or a compatble key in the subtable, returns a failure indication.

The tables are all contained in a file which must be processed to obtain an assembly language program. Thus they are written in a special-purpose language. To provided definiteness to the following discussion, here is an example of a subtable entry.

```
%n,aw
    F
    add   A2,R
```

The '%' indicates the key; the information following (up to a blank line) specifies the code string. Very briefly, this entry is in the subtable for '+' of *regtab;* the key specifies that the left operand is any integer, character, or pointer expression, and the right operand is any word quantity which is directly addressible (e.g. a variable or constant). The code string calls for the generation of the code to compile the left (first) operand into the current register ('F') and then to produce an 'add' instruction which adds the second operand ('A2') to the register ('R'). All of the notation will be explained below.

Only three features of the operands are used in deciding whether a match has occurred. They are:

1. Is the type of the operand compatible with that demanded?
2. Is the 'degree of difficulty' (in a sense described below) compatible?
3. The table may demand that the operand have a '*' (indirection operator) as its highest operator.

As suggested above, the key for a subtable entry is indicated by a '%,' and a comma-separated pair of specifications for the operands. (The second specification is ignored for unary operators). A specification indicates a type requirement by including one of the following letters. If no type letter is present, any integer, character, or pointer operand will satisfy the requirement (not float, double, or long).

b A byte (character) operand is required.

w A word (integer or pointer) operand is required.

f A float or double operand is required.

d A double operand is required.

l A long (32-bit integer) operand is required.

Before discussing the 'degree of difficulty' specification, the algorithm has to be explained more completely. *Rcexpr* (and *cexpr)* are called with a register number in which to place their result. Registers 0, 1, ... are used during evaluation of expressions; the maximum register which can be used in this way depends on the number of register variables, but in any event only registers 0 through 4 are available since r5 is used as a stack frame header and r6 (sp) and r7 (pc) have special hardware properties. The code generation routines assume that when called with register *n* as argument, they may use *n+1,* ... (up to the first register variable) as temporaries. Consider the expression 'X+Y', where both X and Y are expressions. As a first approximation, there are three ways of compiling code to put this expression in register *n*.

1. If Y is an addressible cell, (recursively) put X into register *n* and add Y to it.
2. If Y is an expression that can be calculated in *k* registers, where *k* smaller than the number of registers available, compile X into register *n,* Y into register *n+1,* and add register *n+1* to *n*.
3. Otherwise, compile Y into register *n,* save the result in a temporary (actually, on the stack) compile X into register *n,* then add in the temporary.

The distinction between cases 2 and 3 therefore depends on whether the right operand can be compiled in fewer than *k* registers, where *k* is the number of free registers left after registers 0 through *n* are taken: 0 through *n-1* are presumed to contain already computed temporary results; *n* will, in case 2, contain the value of the left operand while the right is being evaluated.

These considerations should make clear the specification codes for the degree of difficulty, bearing in mind that a number of special cases are also present:

z is satisfied when the operand is zero, so that special code can be produced for expressions like 'x = 0'.

1 is satisfied when the operand is the constant 1, to optimize cases like left and right shift by 1, which can be done efficiently on the PDP-11.

c is satisfied when the operand is a positive (16-bit) constant; this takes care of some special cases in long arithmetic.

a is satisfied when the operand is addressible; this occurs not only for variables and constants, but also for some more complicated constructions, such as indirection through a simple variable, '*p++' where *p* is a register variable (because of the PDP-11's auto-increment address mode), and '*(p+c)' where *p* is a register and *c* is a constant. Precisely, the requirement is that the operand refers to a cell whose address can be written as a source or destination of a PDP-11 instruction.

e is satisfied by an operand whose value can be generated in a register using no more than *k* registers, where *k* is the number of registers left (not counting the current register). The 'e' stands for 'easy.'

n is satisfied by any operand. The 'n' stands for 'anything.'

These degrees of difficulty are considered to lie in a linear ordering and any operand which satisfies an earlier-mentioned requirement will satisfy a later one. Since the subtables are searched linearly, if a '1' specification is included, almost certainly a 'z' must be written first to prevent expressions containing the constant 0 to be compiled as if the 0 were 1.

Finally, a key specification may contain a '*' which requires the operand to have an indirection as its leading operator. Examples below should clarify the utility of this specification.

Now let us consider the contents of the code string associated with each subtable entry. Conventionally, lower-case letters in this string represent literal information which is copied directly to the output. Upper-case letters generally introduce specific macro-operations, some of which may be followed by modifying information. The code strings in the tables are written with tabs and new-lines used freely to suggest instructions which will be generated; the table-compiling program compresses tabs (using the 0200 bit of the next character) and throws away some of the new-lines. For example the macro 'F' is ordinarily written on a line by itself; but since its expansion will end with a new-line, the new-line after 'F' itself is dispensable. This is all to reduce the size of the stored tables.

The first set of macro-operations is concerned with compiling subtrees. Recall that this is done by the *cexpr* routine. In the following discussion the 'current register' is generally the argument register to *cexpr;* that is, the place where the result is desired. The 'next register' is numbered one higher than the current register. (This explanation isn't fully true because of complications, described below, involving operations which require even-odd register pairs.)

F causes a recursive call to the *rcexpr* routine to compile code which places the value of the first (left) operand of the operator in the current register.

F1 generates code which places the value of the first operand in the next register. It is incorrectly used if there might be no next register; that is, if the degree of difficulty of the first operand is not 'easy;' if not, another register might not be available.

FS generates code which pushes the value of the first operand on the stack, by calling *rcexpr* specifying *sptab* as the table.

Analogously,

S, S1, SS
: compile the second (right) operand into the current register, the next register, or onto the stack.

To deal with registers, there are

R which expands into the name of the current register.

R1 which expands into the name of the next register.

R+ which expands into the the name of the current register plus 1. It was suggested above that this is the same as the next register, except for complications; here is one of them. Long integer variables have 32 bits and require 2 registers; in such cases the next register is the current register plus 2. The code would like to talk about both halves of the long quantity, so R refers to the register with the high-order part and R+ to the low-order part.

R– This is another complication, involving division and mod. These operations involve a pair of registers of which the odd-numbered contains the left operand. *Cexpr* arranges that the current register is odd; the R– notation allows the code to refer to the next lower, even-numbered register.

To refer to addressible quantities, there are the notations:

A1 causes generation of the address specified by the first operand. For this to be legal, the operand must be addressible; its key must contain an 'a' or a more restrictive specification.

A2 correspondingly generates the address of the second operand providing it has one.

We now have enough mechanism to show a complete, if suboptimal, table for the + operator on word or byte operands.

```
%n,z
        F

%n,1
        F
        inc   R

%n,aw
        F
        add   A2,R

%n,e
        F
        S1
        add   R1,R

%n,n
        SS
        F
        add   (sp)+,R
```

The first two sequences handle some special cases. Actually it turns out that handling a right operand of 0 is unnecessary since the expression-optimizer throws out adds of 0. Adding 1 by using the 'increment' instruction is done next, and then the case where the right operand is addressible. It must be a word quantity, since the PDP-11 lacks an 'add byte' instruction. Finally the cases where the right operand either can, or cannot, be done in the available registers are treated.

The next macro-instructions are conveniently introduced by noticing that the above table is suitable for subtraction as well as addition, since no use is made of the commutativity of addition. All that is needed is substitution of 'sub' for 'add' and 'dec' for 'inc.' Considerable saving of space

is achieved by factoring out several similar operations.

I is replaced by a string from another table indexed by the operator in the node being expanded. This secondary table actually contains two strings per operator.

I' is replaced by the second string in the side table entry for the current operator.

Thus, given that the entries for '+' and '–' in the side table (which is called *instab)* are 'add' and 'inc,' 'sub' and 'dec' respectively, the middle of of the above addition table can be written

```
%n,1
        F
        I'      R

%n,aw
        F
        I       A2,R
```

and it will be suitable for subtraction, and several other operators, as well.

Next, there is the question of character and floating-point operations.

B1 generates the letter 'b' if the first operand is a character, 'f' if it is float or double, and nothing otherwise. It is used in a context like 'movB1' which generates a 'mov', 'movb', or 'movf' instruction according to the type of the operand.

B2 is just like B1 but applies to the second operand.

BE generates 'b' if either operand is a character and null otherwise.

BF generates 'f' if the type of the operator node itself is float or double, otherwise null.

For example, there is an entry in *efftab* for the '=' operator

```
%a,aw
%ab,a
        IBE   A2,A1
```

Note first that two key specifications can be applied to the same code string. Next, observe that when a word is assigned to a byte or to a word, or a word is assigned to a byte, a single instruction, a *mov* or *movb* as appropriate, does the job. However, when a byte is assigned to a word, it must pass through a register to implement the sign-extension rules:

```
%a,n
        S
        IB1   R,A1
```

Next, there is the question of handling indirection properly. Consider the expression 'X + *Y', where X and Y are expressions, Assuming that Y is more complicated than just a variable, but on the other hand qualifies as 'easy' in the context, the expression would be compiled by placing the value of X in a register, that of *Y in the next register, and adding the registers. It is easy to see that a better job can be done by compiling X, then Y (into the next register), and producing the instruction symbolized by 'add (R1),R'. This scheme avoids generating the instruction 'mov (R1),R1' required actually to place the value of *Y in a register. A related situation occurs with the expression 'X + *(p+6)', which exemplifies a construction frequent in structure and array references. The addition table shown above would produce

```
[put X in register R]
mov  p,R1
add  $6,R1
mov  (R1),R1
add  R1,R
```

when the best code is

```
[put X in R]
mov  p,R1
add  6(R1),R
```

As we said above, a key specification for a code table entry may require an operand to have an indirection as its highest operator. To make use of the requirement, the following macros are provided.

F* the first operand must have the form *X. If in particular it has the form *(Y + c), for some constant *c*, then code is produced which places the value of Y in the current register. Otherwise, code is produced which loads X into the current register.

F1* resembles F* except that the next register is loaded.

S* resembles F* except that the second operand is loaded.

S1* resembles S* except that the next register is loaded.

FS* The first operand must have the form '*X'. Push the value of X on the stack.

SS* resembles FS* except that it applies to the second operand.

To capture the constant that may have been skipped over in the above macros, there are

#1 The first operand must have the form *X; if in particular it has the form *(Y + c) for *c* a constant, then the constant is written out, otherwise a null string.

#2 is the same as #1 except that the second operand is used.

Now we can improve the addition table above. Just before the '%n,e' entry, put

```
%n,ew*
        F
        S1*
        add  #2(R1),R
```

and just before the '%n,n' put

```
%n,nw*
        SS*
        F
        add  *(sp)+,R
```

When using the stacking macros there is no place to use the constant as an index word, so that particular special case doesn't occur.

The constant mentioned above can actually be more general than a number. Any quantity acceptable to the assembler as an expression will do, in particular the address of a static cell, perhaps with a numeric offset. If *x* is an external character array, the expression 'x[i+5] = 0' will generate the code

```
mov  i,r0
clrb x+5(r0)
```

via the table entry (in the '=' part of *efftab)*

```
%e*,z
        F
        I'B1 #1(R)
```

Some machine operations place restrictions on the registers used. The divide instruction, used to implement the divide and mod operations, requires the dividend to be placed in the odd member of an even-odd pair; other peculiarities of multiplication make it simplest to put the multiplicand in an odd-numbered register. There is no theory which optimally accounts for this kind of requirement. *Cexpr* handles it by checking for a multiply, divide, or mod operation; in these cases, its argument register number is incremented by one or two so that it is odd, and if the operation was

divide or mod, so that it is a member of a free even-odd pair. The routine which determines the number of registers required estimates, conservatively, that at least two registers are required for a multiplication and three for the other peculiar operators. After the expression is compiled, the register where the result actually ended up is returned. (Divide and mod are actually the same operation except for the location of the result).

These operations are the ones which cause results to end up in unexpected places, and this possibility adds a further level of complexity. The simplest way of handling the problem is always to move the result to the place where the caller expected it, but this will produce unnecessary register moves in many simple cases; 'a = b*c' would generate

```
mov  b,r1
mul  c,r1
mov  r1,r0
mov  r0,a
```

The next thought is used the passed-back information as to where the result landed to change the notion of the current register. While compiling the '=' operation above, which comes from a table entry like

```
%a,e
        S
        mov  R,A1
```

it is sufficient to redefine the meaning of 'R' after processing the 'S' which does the multiply. This technique is in fact used; the tables are written in such a way that correct code is produced. The trouble is that the technique cannot be used in general, because it invalidates the count of the number of registers required for an expression. Consider just 'a*b + X' where X is some expression. The algorithm assumes that the value of a*b, once computed, requires just one register. If there are three registers available, and X requires two registers to compute, then this expression will match a key specifying '%n,e'. If a*b is computed and left in register 1, then there are, contrary to expectations, no longer two registers available to compute X, but only one, and bad code will be produced. To guard against this possibility, *cexpr* checks the result returned by recursive calls which implement F, S and their relatives. If the result is not in the expected register, then the number of registers required by the other operand is checked; if it can be done using those registers which remain even after making unavailable the unexpectedly-occupied register, then the notions of the 'next register' and possibly the 'current register' are redefined. Otherwise a register-copy instruction is produced. A register-copy is also always produced when the current operator is one of those which have odd-even requirements.

Finally, there are a few loose-end macro operations and facts about the tables. The operators:

V is used for long operations. It is written with an address like a machine instruction; it expands into 'adc' (add carry) if the operation is an additive operator, 'sbc' (subtract carry) if the operation is a subtractive operator, and disappears, along with the rest of the line, otherwise. Its purpose is to allow common treatment of logical operations, which have no carries, and additive and subtractive operations, which generate carries.

T generates a 'tst' instruction if the first operand of the tree does not set the condition codes correctly. It is used with divide and mod operations, which require a sign-extended 32-bit operand. The code table for the operations contains an 'sxt' (sign-extend) instruction to generate the high-order part of the dividend.

H is analogous to the 'F' and 'S' macros, except that it calls for the generation of code for the current tree (not one of its operands) using *regtab*. It is used in *cctab* for all the operators which, when executed normally, set the condition codes properly according to the result. It prevents a 'tst' instruction from being generated for constructions like 'if (a+b) ...' since after calculation of the value of 'a+b' a conditional branch can be written immediately.

All of the discussion above is in terms of operators with operands. Leaves of the expression tree (variables and constants), however, are peculiar in that they have no operands. In order to regularize the matching process, *cexpr* examines its operand to determine if it is a leaf; if so, it creates a special 'load' operator whose operand is the leaf, and substitutes it for the argument tree; this allows the table entry for the created operator to use the 'A1' notation to load the leaf into a register.

Purely to save space in the tables, pieces of subtables can be labelled and referred to later. It turns out, for example, that rather large portions of the the *efftab* table for the '=' and '=+' operators are identical. Thus '=' has an entry

```
%[move3:]
%a,aw
%ab,a
        IBE  A2,A1
```

while part of the '=+' table is

```
%aw,aw
%    [move3]
```

Labels are written as '%[... :]', before the key specifications; references are written with '% [...]' after the key. Peculiarities in the implementation make it necessary that labels appear before references to them.

The example illustrates the utility of allowing separate keys to point to the same code string. The assignment code works properly if either the right operand is a word, or the left operand is a byte; but since there is no 'add byte' instruction the addition code has to be restricted to word operands.

Delaying and reordering

Intertwined with the code generation routines are two other, interrelated processes. The first, implemented by a routine called *delay*, is based on the observation that naive code generation for the expression 'a = b++' would produce

```
mov  b,r0
inc  b
mov  r0,a
```

The point is that the table for postfix ++ has to preserve the value of *b* before incrementing it; the general way to do this is to preserve its value in a register. A cleverer scheme would generate

```
mov  b,a
inc  b
```

Delay is called for each expression input to *rcexpr*, and it searches for postfix ++ and — operators. If one is found applied to a variable, the tree is patched to bypass the operator and compiled as it stands; then the increment or decrement itself is done. The effect is as if 'a = b; b++' had been written. In this example, of course, the user himself could have done the same job, but more complicated examples are easily constructed, for example 'switch (x++)'. An essential restriction is that the condition codes not be required. It would be incorrect to compile 'if (a++) ...' as

```
tst  a
inc  a
beq  ...
```

because the 'inc' destroys the required setting of the condition codes.

Reordering is a similar sort of optimization. Many cases which it detects are useful mainly with register variables. If *r* is a register variable, the expression 'r = x+y' is best compiled as

```
mov  x,r
add  y,r
```

but the codes tables would produce

```
mov  x,r0
add  y,r0
mov  r0,r
```

which is in fact preferred if *r* is not a register. (If *r* is not a register, the two sequences are the same size, but the second is slightly faster.) The scheme is to compile the expression as if it had been written 'r = x; r =+ y'. The *reorder* routine is called with a pointer to each tree that *rcexpr* is about to compile; if it has the right characteristics, the 'r = x' tree is constructed and passed recursively to *rcexpr;* then the original tree is modified to read 'r =+ y' and the calling instance of *rcexpr* compiles that instead. Of course the whole business is itself recursive so that more extended forms of the same phenomenon are handled, like 'r = x + y | z'.

Care does have to be taken to avoid 'optimizing' an expression like 'r = x + r' into 'r = x; r =+ r'. It is required that the right operand of the expression on the right of the '=' be a ', distinct from the register variable.

The second case that *reorder* handles is expressions of the form 'r = X' used as a subexpression. Again, the code out of the tables for 'x = r = y' would be

```
mov  y,r0
mov  r0,r
mov  r0,x
```

whereas if *r* were a register it would be better to produce

```
mov  y,r
mov  r,x
```

When *reorder* discovers that a register variable is being assigned to in a subexpression, it calls *rcexpr* recursively to compile the subexpression, then fiddles the tree passed to it so that the register variable itself appears as the operand instead of the whole subexpression. Here care has to be taken to avoid an infinite regress, with *rcexpr* and *reorder* calling each other forever to handle assignments to registers.

A third set of cases treated by *reorder* comes up when any name, not necessarily a register, occurs as a left operand of an assignment operator other than '=' or as an operand of prefix '++' or '—'. Unless condition-code tests are involved, when a subexpression like '(a =+ b)' is seen, the assignment is performed and the argument tree modified so that *a* is its operand; effectively 'x + (y =+ z)' is compiled as 'y =+ z; x + y'. Similarly, prefix increment and decrement are pulled out and performed first, then the remainder of the expression.

Throughout code generation, the expression optimizer is called whenever *delay* or *reorder* change the expression tree. This allows some special cases to be found that otherwise would not be seen.

A Tour Through the Portable C Compiler

S. C. Johnson

Introduction

A C compiler has been implemented that has proved to be quite portable, serving as the basis for C compilers on roughly a dozen machines, including the Honeywell 6000, IBM 370, and Interdata 8/32. The compiler is highly compatible with the C language standard.[1]

Among the goals of this compiler are portability, high reliability, and the use of state-of-the-art techniques and tools wherever practical. Although the efficiency of the compiling process is not a primary goal, the compiler is efficient enough, and produces good enough code, to serve as a production compiler.

The language implemented is highly compatible with the current PDP-11 version of C. Moreover, roughly 75% of the compiler, including nearly all the syntactic and semantic routines, is machine independent. The compiler also serves as the major portion of the program *lint*, described elsewhere.[2]

A number of earlier attempts to make portable compilers are worth noting. While on CO-OP assignment to Bell Labs in 1973, Alan Snyder wrote a portable C compiler which was the basis of his Master's Thesis at M.I.T.[3] This compiler was very slow and complicated, and contained a number of rather serious implementation difficulties; nevertheless, a number of Snyder's ideas appear in this work.

Most earlier portable compilers, including Snyder's, have proceeded by defining an intermediate language, perhaps based on three-address code or code for a stack machine, and writing a machine independent program to translate from the source code to this intermediate code. The intermediate code is then read by a second pass, and interpreted or compiled. This approach is elegant, and has a number of advantages, especially if the target machine is far removed from the host. It suffers from some disadvantages as well. Some constructions, like initialization and subroutine prologs, are difficult or expensive to express in a machine independent way that still allows them to be easily adapted to the target assemblers. Most of these approaches require a symbol table to be constructed in the second (machine dependent) pass, and/or require powerful target assemblers. Also, many conversion operators may be generated that have no effect on a given machine, but may be needed on others (for example, pointer to pointer conversions usually do nothing in C, but must be generated because there are some machines where they are significant).

For these reasons, the first pass of the portable compiler is not entirely machine independent. It contains some machine dependent features, such as initialization, subroutine prolog and epilog, certain storage allocation functions, code for the *switch* statement, and code to throw out unneeded conversion operators.

As a crude measure of the degree of portability actually achieved, the Interdata 8/32 C compiler has roughly 600 machine dependent lines of source out of 4600 in Pass 1, and 1000 out of 3400 in Pass 2. In total, 1600 out of 8000, or 20%, of the total source is machine dependent (12% in Pass 1, 30% in Pass 2). These percentages can be expected to rise slightly as the compiler is tuned. The percentage of machine-dependent code for the IBM is 22%, for the Honeywell 25%. If the assembler format and structure were the same for all these machines, perhaps another 5-10% of the code would become machine independent.

These figures are sufficiently misleading as to be almost meaningless. A large fraction of the machine dependent code can be converted in a straightforward, almost mechanical way. On the

other hand, a certain amount of the code requres hard intellectual effort to convert, since the algorithms embodied in this part of the code are typically complicated and machine dependent.

To summarize, however, if you need a C compiler written for a machine with a reasonable architecture, the compiler is already three quarters finished!

Overview

This paper discusses the structure and organization of the portable compiler. The intent is to give the big picture, rather than discussing the details of a particular machine implementation. After a brief overview and a discussion of the source file structure, the paper describes the major data structures, and then delves more closely into the two passes. Some of the theoretical work on which the compiler is based, and its application to the compiler, is discussed elsewhere.[4] One of the major design issues in any C compiler, the design of the calling sequence and stack frame, is the subject of a separate memorandum.[5]

The compiler consists of two passes, *pass1* and *pass2*, that together turn C source code into assembler code for the target machine. The two passes are preceded by a preprocessor, that handles the **#define** and **#include** statements, and related features (e.g., **#ifdef**, etc.). It is a nearly machine independent program, and will not be further discussed here.

The output of the preprocessor is a text file that is read as the standard input of the first pass. This produces as standard output another text file that becomes the standard input of the second pass. The second pass produces, as standard output, the desired assembler language source code. The preprocessor and the two passes all write error messages on the standard error file. Thus the compiler itself makes few demands on the I/O library support, aiding in the bootstrapping process.

Although the compiler is divided into two passes, this represents historical accident more than deep necessity. In fact, the compiler can optionally be loaded so that both passes operate in the same program. This "one pass" operation eliminates the overhead of reading and writing the intermediate file, so the compiler operates about 30% faster in this mode. It also occupies about 30% more space than the larger of the two component passes.

Because the compiler is fundamentally structured as two passes, even when loaded as one, this document primarily describes the two pass version.

The first pass does the lexical analysis, parsing, and symbol table maintenance. It also constructs parse trees for expressions, and keeps track of the types of the nodes in these trees. Additional code is devoted to initialization. Machine dependent portions of the first pass serve to generate subroutine prologs and epilogs, code for switches, and code for branches, label definitions, alignment operations, changes of location counter, etc.

The intermediate file is a text file organized into lines. Lines beginning with a right parenthesis are copied by the second pass directly to its output file, with the parenthesis stripped off. Thus, when the first pass produces assembly code, such as subroutine prologs, etc., each line is prefaced with a right parenthesis; the second pass passes these lines to through to the assembler.

The major job done by the second pass is generation of code for expressions. The expression parse trees produced in the first pass are written onto the intermediate file in Polish Prefix form: first, there is a line beginning with a period, followed by the source file line number and name on which the expression appeared (for debugging purposes). The successive lines represent the nodes of the parse tree, one node per line. Each line contains the node number, type, and any values (e.g., values of constants) that may appear in the node. Lines representing nodes with descendants are immediately followed by the left subtree of descendants, then the right. Since the number of descendants of any node is completely determined by the node number, there is no need to mark the end of the tree.

There are only two other line types in the intermediate file. Lines beginning with a left square bracket ('[') represent the beginning of blocks (delimited by { ... } in the C source); lines beginning with right square brackets (']') represent the end of blocks. The remainder of these lines tell how much stack space, and how many register variables, are currently in use.

Thus, the second pass reads the intermediate files, copies the ')' lines, makes note of the information in the '[' and ']' lines, and devotes most of its effort to the '.' lines and their associated expression trees, turning them turns into assembly code to evaluate the expressions.

In the one pass version of the compiler, the expression trees that are built by the first pass have been declared to have room for the second pass information as well. Instead of writing the trees onto an intermediate file, each tree is transformed in place into an acceptable form for the code generator. The code generator then writes the result of compiling this tree onto the standard output. Instead of '[' and ']' lines in the intermediate file, the information is passed directly to the second pass routines. Assembly code produced by the first pass is simply written out, without the need for ')' at the head of each line.

The Source Files

The compiler source consists of 22 source files. Two files, *manifest* and *macdefs*, are header files included with all other files. *Manifest* has declarations for the node numbers, types, storage classes, and other global data definitions. *Macdefs* has machine-dependent definitions, such as the size and alignment of the various data representations. Two machine independent header files, *mfile1* and *mfile2*, contain the data structure and manifest definitions for the first and second passes, respectively. In the second pass, a machine dependent header file, *mac2defs*, contains declarations of register names, etc.

There is a file, *common*, containing (machine independent) routines used in both passes. These include routines for allocating and freeing trees, walking over trees, printing debugging information, and printing error messages. There are two dummy files, *comm1.c* and *comm2.c*, that simply include *common* within the scope of the appropriate pass1 or pass2 header files. When the compiler is loaded as a single pass, *common* only needs to be included once: *comm2.c* is not needed.

Entire sections of this document are devoted to the detailed structure of the passes. For the moment, we just give a brief description of the files. The first pass is obtained by compiling and loading *scan.c*, *cgram.c*, *xdefs.c*, *pftn.c*, *trees.c*, *optim.c*, *local.c*, *code.c*, and *comm1.c*. *Scan.c* is the lexical analyzer, which is used by *cgram.c*, the result of applying *Yacc*[6] to the input grammar *cgram.y*. *Xdefs.c* is a short file of external definitions. *Pftn.c* maintains the symbol table, and does initialization. *Trees.c* builds the expression trees, and computes the node types. *Optim.c* does some machine independent optimizations on the expression trees. *Comm1.c* includes *common*, that contains service routines common to the two passes of the compiler. All the above files are machine independent. The files *local.c* and *code.c* contain machine dependent code for generating subroutine prologs, switch code, and the like.

The second pass is produced by compiling and loading *reader.c*, *allo.c*, *match.c*, *comm1.c*, *order.c*, *local.c*, and *table.c*. *Reader.c* reads the intermediate file, and controls the major logic of the code generation. *Allo.c* keeps track of busy and free registers. *Match.c* controls the matching of code templates to subtrees of the expression tree to be compiled. *Comm2.c* includes the file *common*, as in the first pass. The above files are machine independent. *Order.c* controls the machine dependent details of the code generation strategy. *Local2.c* has many small machine dependent routines, and tables of opcodes, register types, etc. *Table.c* has the code template tables, which are also clearly machine dependent.

Data Structure Considerations.

This section discusses the node numbers, type words, and expression trees, used throughout both passes of the compiler.

The file *manifest* defines those symbols used throughout both passes. The intent is to use the same symbol name (e.g., MINUS) for the given operator throughout the lexical analysis, parsing, tree building, and code generation phases; this requires some synchronization with the *Yacc* input file, *cgram.y*, as well.

A token like MINUS may be seen in the lexical analyzer before it is known whether it is a unary or binary operator; clearly, it is necessary to know this by the time the parse tree is constructed. Thus, an operator (really a macro) called UNARY is provided, so that MINUS and UNARY MINUS are both distinct node numbers. Similarly, many binary operators exist in an assignment form (for example, -=), and the operator ASG may be applied to such node names to generate new ones, e.g. ASG MINUS.

It is frequently desirable to know if a node represents a leaf (no descendants), a unary operator (one descendant) or a binary operator (two descendants). The macro *optype(o)* returns one of the manifest constants LTYPE, UTYPE, or BITYPE, respectively, depending on the node number *o*. Similarly, *asgop(o)* returns true if *o* is an assignment operator number (=, +=, etc.), and *logop(o)* returns true if *o* is a relational or logical (&&, || or !) operator.

C has a rich typing structure, with a potentially infinite number of types. To begin with, there are the basic types: CHAR, SHORT, INT, LONG, the unsigned versions known as UCHAR, USHORT, UNSIGNED, ULONG, and FLOAT, DOUBLE, and finally STRTY (a structure), UNIONTY, and ENUMTY. Then, there are three operators that can be applied to types to make others: if *t* is a type, we may potentially have types *pointer to t*, *function returning t*, and *array of t's* generated from *t*. Thus, an arbitrary type in C consists of a basic type, and zero or more of these operators.

In the compiler, a type is represented by an unsigned integer; the rightmost four bits hold the basic type, and the remaining bits are divided into two-bit fields, containing 0 (no operator), or one of the three operators described above. The modifiers are read right to left in the word, starting with the two-bit field adjacent to the basic type, until a field with 0 in it is reached. The macros PTR, FTN, and ARY represent the *pointer to*, *function returning*, and *array of* operators. The macro values are shifted so that they align with the first two-bit field; thus PTR+INT represents the type for an integer pointer, and

ARY + (PTR<<2) + (FTN<<4) + DOUBLE

represents the type of an array of pointers to functions returning doubles.

The type words are ordinarily manipulated by macros. If *t* is a type word, *BTYPE(t)* gives the basic type. *ISPTR(t)*, *ISARY(t)*, and *ISFTN(t)* ask if an object of this type is a pointer, array, or a function, respectively. *MODTYPE(t,b)* sets the basic type of *t* to *b*. *DECREF(t)* gives the type resulting from removing the first operator from *t*. Thus, if *t* is a pointer to *t'*, a function returning *t'*, or an array of *t'*, then *DECREF(t)* would equal *t'*. *INCREF(t)* gives the type representing a pointer to *t*. Finally, there are operators for dealing with the unsigned types. *ISUNSIGNED(t)* returns true if *t* is one of the four basic unsigned types; in this case, *DEUNSIGN(t)* gives the associated 'signed' type. Similarly, *UNSIGNABLE(t)* returns true if *t* is one of the four basic types that could become unsigned, and *ENUNSIGN(t)* returns the unsigned analogue of *t* in this case.

The other important global data structure is that of expression trees. The actual shapes of the nodes are given in *mfile1* and *mfile2*. They are not the same in the two passes; the first pass nodes contain dimension and size information, while the second pass nodes contain register allocation information. Nevertheless, all nodes contain fields called *op*, containing the node number, and *type*, containing the type word. A function called *talloc()* returns a pointer to a new tree node. To free a node, its *op* field need merely be set to FREE. The other fields in the node will remain intact at least until the next allocation.

Nodes representing binary operators contain fields, *left* and *right*, that contain pointers to the left and right descendants. Unary operator nodes have the *left* field, and a value field called *rval*. Leaf nodes, with no descendants, have two value fields: *lval* and *rval*.

At appropriate times, the function *tcheck()* can be called, to check that there are no busy nodes remaining. This is used as a compiler consistency check. The function *tcopy(p)* takes a pointer *p* that points to an expression tree, and returns a pointer to a disjoint copy of the tree. The function *walkf(p,f)* performs a postorder walk of the tree pointed to by *p*, and applies the

function *f* to each node. The function *fwalk(p,f,d)* does a preorder walk of the tree pointed to by *p*. At each node, it calls a function *f*, passing to it the node pointer, a value passed down from its ancestor, and two pointers to values to be passed down to the left and right descendants (if any). The value *d* is the value passed down to the root. *Fwalk* is used for a number of tree labeling and debugging activities.

The other major data structure, the symbol table, exists only in pass one, and will be discussed later.

Pass One

The first pass does lexical analysis, parsing, symbol table maintenance, tree building, optimization, and a number of machine dependent things. This pass is largely machine independent, and the machine independent sections can be pretty successfully ignored. Thus, they will be only sketched here.

Lexical Analysis

The lexical analyzer is a conceptually simple routine that reads the input and returns the tokens of the C language as it encounters them: names, constants, operators, and keywords. The conceptual simplicity of this job is confounded a bit by several other simple jobs that unfortunately must go on simultaneously. These include

- Keeping track of the current filename and line number, and occasionally setting this information as the result of preprocessor control lines.
- Skipping comments.
- Properly dealing with octal, decimal, hex, floating point, and character constants, as well as character strings.

To achieve speed, the program maintains several tables that are indexed into by character value, to tell the lexical analyzer what to do next. To achieve portability, these tables must be initialized each time the compiler is run, in order that the table entries reflect the local character set values.

Parsing

As mentioned above, the parser is generated by Yacc from the grammar on file *cgram.y*. The grammar is relatively readable, but contains some unusual features that are worth comment.

Perhaps the strangest feature of the grammar is the treatment of declarations. The problem is to keep track of the basic type and the storage class while interpreting the various stars, brackets, and parentheses that may surround a given name. The entire declaration mechanism must be recursive, since declarations may appear within declarations of structures and unions, or even within a **sizeof** construction inside a dimension in another declaration!

There are some difficulties in using a bottom-up parser, such as produced by Yacc, to handle constructions where a lot of left context information must be kept around. The problem is that the original PDP-11 compiler is top-down in implementation, and some of the semantics of C reflect this. In a top-down parser, the input rules are restricted somewhat, but one can naturally associate temporary storage with a rule at a very early stage in the recognition of that rule. In a bottom-up parser, there is more freedom in the specification of rules, but it is more difficult to know what rule is being matched until the entire rule is seen. The parser described by *cgram.c* makes effective use of the bottom-up parsing mechanism in some places (notably the treatment of expressions), but struggles against the restrictions in others. The usual result is that it is necessary to run a stack of values "on the side", independent of the Yacc value stack, in order to be able to store and access information deep within inner constructions, where the relationship of the rules being recognized to the total picture is not yet clear.

In the case of declarations, the attribute information (type, etc.) for a declaration is carefully kept immediately to the left of the declarator (that part of the declaration involving the name). In

this way, when it is time to declare the name, the name and the type information can be quickly brought together. The "$0" mechanism of Yacc is used to accomplish this. The result is not pretty, but it works. The storage class information changes more slowly, so it is kept in an external variable, and stacked if necessary. Some of the grammar could be considerably cleaned up by using some more recent features of Yacc, notably actions within rules and the ability to return multiple values for actions.

A stack is also used to keep track of the current location to be branched to when a **break** or **continue** statement is processed.

This use of external stacks dates from the time when Yacc did not permit values to be structures. Some, or most, of this use of external stacks could be eliminated by redoing the grammar to use the mechanisms now provided. There are some areas, however, particularly the processing of structure, union, and enum declarations, function prologs, and switch statement processing, when having all the affected data together in an array speeds later processing; in this case, use of external storage seems essential.

The *cgram.y* file also contains some small functions used as utility functions in the parser. These include routines for saving case values and labels in processing switches, and stacking and popping values on the external stack described above.

Storage Classes

C has a finite, but fairly extensive, number of storage classes available. One of the compiler design decisions was to process the storage class information totally in the first pass; by the second pass, this information must have been totally dealt with. This means that all of the storage allocation must take place in the first pass, so that references to automatics and parameters can be turned into references to cells lying a certain number of bytes offset from certain machine registers. Much of this transformation is machine dependent, and strongly depends on the storage class.

The classes include EXTERN (for externally declared, but not defined variables), EXTDEF (for external definitions), and similar distinctions for USTATIC and STATIC, UFORTRAN and FORTRAN (for fortran functions) and ULABEL and LABEL. The storage classes REGISTER and AUTO are obvious, as are STNAME, UNAME, and ENAME (for structure, union, and enumeration tags), and the associated MOS, MOU, and MOE (for the members). TYPEDEF is treated as a storage class as well. There are two special storage classes: PARAM and SNULL. SNULL is used to distinguish the case where no explicit storage class has been given; before an entry is made in the symbol table the true storage class is discovered. Similarly, PARAM is used for the temporary entry in the symbol table made before the declaration of function parameters is completed.

The most complexity in the storage class process comes from bit fields. A separate storage class is kept for each width bit field; a *k* bit bit field has storage class *k* plus FIELD. This enables the size to be quickly recovered from the storage class.

Symbol Table Maintenance.

The symbol table routines do far more than simply enter names into the symbol table; considerable semantic processing and checking is done as well. For example, if a new declaration comes in, it must be checked to see if there is a previous declaration of the same symbol. If there is, there are many cases. The declarations may agree and be compatible (for example, an extern declaration can appear twice) in which case the new declaration is ignored. The new declaration may add information (such as an explicit array dimension) to an already present declaration. The new declaration may be different, but still correct (for example, an extern declaration of something may be entered, and then later the definition may be seen). The new declaration may be incompatible, but appear in an inner block; in this case, the old declaration is carefully hidden away, and the new one comes into force until the block is left. Finally, the declarations may be incompatible, and an error message must be produced.

A number of other factors make for additional complexity. The type declared by the user is not always the type entered into the symbol table (for example, if an formal parameter to a

function is declared to be an array, C requires that this be changed into a pointer before entry in the symbol table). Moreover, there are various kinds of illegal types that may be declared which are difficult to check for syntactically (for example, a function returning an array). Finally, there is a strange feature in C that requires structure tag names and member names for structures and unions to be taken from a different logical symbol table than ordinary identifiers. Keeping track of which kind of name is involved is a bit of struggle (consider typedef names used within structure declarations, for example).

The symbol table handling routines have been rewritten a number of times to extend features, improve performance, and fix bugs. They address the above problems with reasonable effectiveness but a singular lack of grace.

When a name is read in the input, it is hashed, and the routine *lookup* is called, together with a flag which tells which symbol table should be searched (actually, both symbol tables are stored in one, and a flag is used to distinguish individual entries). If the name is found, *lookup* returns the index to the entry found; otherwise, it makes a new entry, marks it UNDEF (undefined), and returns the index of the new entry. This index is stored in the *rval* field of a NAME node.

When a declaration is being parsed, this NAME node is made part of a tree with UNARY MUL nodes for each *, LB nodes for each array descriptor (the right descendant has the dimension), and UNARY CALL nodes for each function descriptor. This tree is passed to the routine *tymerge*, along with the attribute type of the whole declaration; this routine collapses the tree to a single node, by calling *tyreduce*, and then modifies the type to reflect the overall type of the declaration.

Dimension and size information is stored in a table called *dimtab*. To properly describe a type in C, one needs not just the type information but also size information (for structures and enums) and dimension information (for arrays). Sizes and offsets are dealt with in the compiler by giving the associated indices into *dimtab*. *Tymerge* and *tyreduce* call *dstash* to put the discovered dimensions away into the *dimtab* array. *Tymerge* returns a pointer to a single node that contains the symbol table index in its *rval* field, and the size and dimension indices in fields *csiz* and *cdim*, respectively. This information is properly considered part of the type in the first pass, and is carried around at all times.

To enter an element into the symbol table, the routine *defid* is called; it is handed a storage class, and a pointer to the node produced by *tymerge*. *Defid* calls *fixtype*, which adjusts and checks the given type depending on the storage class, and converts null types appropriately. It then calls *fixclass*, which does a similar job for the storage class; it is here, for example, that register declarations are either allowed or changed to auto.

The new declaration is now compared against an older one, if present, and several pages of validity checks performed. If the definitions are compatible, with possibly some added information, the processing is straightforward. If the definitions differ, the block levels of the current and the old declaration are compared. The current block level is kept in *blevel*, an external variable; the old declaration level is kept in the symbol table. Block level 0 is for external declarations, 1 is for arguments to functions, and 2 and above are blocks within a function. If the current block level is the same as the old declaration, an error results. If the current block level is higher, the new declaration overrides the old. This is done by marking the old symbol table entry "hidden", and making a new entry, marked "hiding". *Lookup* will skip over hidden entries. When a block is left, the symbol table is searched, and any entries defined in that block are destroyed; if they hid other entries, the old entries are "unhidden".

This nice block structure is warped a bit because labels do not follow the block structure rules (one can do a **goto** into a block, for example); default definitions of functions in inner blocks also persist clear out to the outermost scope. This implies that cleaning up the symbol table after block exit is more subtle than it might first seem.

For successful new definitions, *defid* also initializes a "general purpose" field, *offset*, in the symbol table. It contains the stack offset for automatics and parameters, the register number for

register variables, the bit offset into the structure for structure members, and the internal label number for static variables and labels. The offset field is set by *falloc* for bit fields, and *dclstruct* for structures and unions.

The symbol table entry itself thus contains the name, type word, size and dimension offsets, offset value, and declaration block level. It also has a field of flags, describing what symbol table the name is in, and whether the entry is hidden, or hides another. Finally, a field gives the line number of the last use, or of the definition, of the name. This is used mainly for diagnostics, but is useful to *lint* as well.

In some special cases, there is more than the above amount of information kept for the use of the compiler. This is especially true with structures; for use in initialization, structure declarations must have access to a list of the members of the structure. This list is also kept in *dimtab*. Because a structure can be mentioned long before the members are known, it is necessary to have another level of indirection in the table. The two words following the *csiz* entry in *dimtab* are used to hold the alignment of the structure, and the index in dimtab of the list of members. This list contains the symbol table indices for the structure members, terminated by a -1.

Tree Building

The portable compiler transforms expressions into expression trees. As the parser recognizes each rule making up an expression, it calls *buildtree* which is given an operator number, and pointers to the left and right descendants. *Buildtree* first examines the left and right descendants, and, if they are both constants, and the operator is appropriate, simply does the constant computation at compile time, and returns the result as a constant. Otherwise, *buildtree* allocates a node for the head of the tree, attaches the descendants to it, and ensures that conversion operators are generated if needed, and that the type of the new node is consistent with the types of the operands. There is also a considerable amount of semantic complexity here; many combinations of types are illegal, and the portable compiler makes a strong effort to check the legality of expression types completely. This is done both for *lint* purposes, and to prevent such semantic errors from being passed through to the code generator.

The heart of *buildtree* is a large table, accessed by the routine *opact*. This routine maps the types of the left and right operands into a rather smaller set of descriptors, and then accesses a table (actually encoded in a switch statement) which for each operator and pair of types causes an action to be returned. The actions are logical or's of a number of separate actions, which may be carried out by *buildtree*. These component actions may include checking the left side to ensure that it is an lvalue (can be stored into), applying a type conversion to the left or right operand, setting the type of the new node to the type of the left or right operand, calling various routines to balance the types of the left and right operands, and suppressing the ordinary conversion of arrays and function operands to pointers. An important operation is OTHER, which causes some special code to be invoked in *buildtree*, to handle issues which are unique to a particular operator. Examples of this are structure and union reference (actually handled by the routine *stref*), the building of NAME, ICON, STRING and FCON (floating point constant) nodes, unary * and &, structure assignment, and calls. In the case of unary * and &, *buildtree* will cancel a * applied to a tree, the top node of which is &, and conversely.

Another special operation is PUN; this causes the compiler to check for type mismatches, such as intermixing pointers and integers.

The treatment of conversion operators is still a rather strange area of the compiler (and of C!). The recent introduction of type casts has only confounded this situation. Most of the conversion operators are generated by calls to *tymatch* and *ptmatch*, both of which are given a tree, and asked to make the operands agree in type. *Ptmatch* treats the case where one of the operands is a pointer; *tymatch* treats all other cases. Where these routines have decided on the proper type for an operand, they call *makety*, which is handed a tree, and a type word, dimension offset, and size offset. If necessary, it inserts a conversion operation to make the types correct. Conversion operations are never inserted on the left side of assignment operators, however. There are two conversion operators used; PCONV, if the conversion is to a non-basic type (usually a pointer), and

SCONV, if the conversion is to a basic type (scalar).

To allow for maximum flexibility, every node produced by ***buildtree*** is given to a machine dependent routine, ***clocal***, immediately after it is produced. This is to allow more or less immediate rewriting of those nodes which must be adapted for the local machine. The conversion operations are given to ***clocal*** as well; on most machines, many of these conversions do nothing, and should be thrown away (being careful to retain the type). If this operation is done too early, however, later calls to ***buildtree*** may get confused about correct type of the subtrees; thus ***clocal*** is given the conversion ops only after the entire tree is built. This topic will be dealt with in more detail later.

Initialization

Initialization is one of the messier areas in the portable compiler. The only consolation is that most of the mess takes place in the machine independent part, where it is may be safely ignored by the implementor of the compiler for a particular machine.

The basic problem is that the semantics of initialization really calls for a co-routine structure; one collection of programs reading constants from the input stream, while another, independent set of programs places these constants into the appropriate spots in memory. The dramatic differences in the local assemblers also come to the fore here. The parsing problems are dealt with by keeping a rather extensive stack containing the current state of the initialization; the assembler problems are dealt with by having a fair number of machine dependent routines.

The stack contains the symbol table number, type, dimension index, and size index for the current identifier being initialized. Another entry has the offset, in bits, of the beginning of the current identifier. Another entry keeps track of how many elements have been seen, if the current identifier is an array. Still another entry keeps track of the current member of a structure being initialized. Finally, there is an entry containing flags which keep track of the current state of the initialization process (e.g., tell if a } has been seen for the current identifier.)

When an initialization begins, the routine ***beginit*** is called; it handles the alignment restrictions, if any, and calls ***instk*** to create the stack entry. This is done by first making an entry on the top of the stack for the item being initialized. If the top entry is an array, another entry is made on the stack for the first element. If the top entry is a structure, another entry is made on the stack for the first member of the structure. This continues until the top element of the stack is a scalar. *Instk* then returns, and the parser begins collecting initializers.

When a constant is obtained, the routine ***doinit*** is called; it examines the stack, and does whatever is necessary to assign the current constant to the scalar on the top of the stack. ***gotscal*** is then called, which rearranges the stack so that the next scalar to be initialized gets placed on top of the stack. This process continues until the end of the initializers; ***endinit*** cleans up. If a { or } is encountered in the string of initializers, it is handled by calling ***ilbrace*** or ***irbrace***, respectively.

A central issue is the treatment of the "holes" that arise as a result of alignment restrictions or explicit requests for holes in bit fields. There is a global variable, ***inoff***, which contains the current offset in the initialization (all offsets in the first pass of the compiler are in bits). *Doinit* figures out from the top entry on the stack the expected bit offset of the next identifier; it calls the machine dependent routine ***inforce*** which, in a machine dependent way, forces the assembler to set aside space if need be so that the next scalar seen will go into the appropriate bit offset position. The scalar itself is passed to one of the machine dependent routines ***fincode*** (for floating point initialization), ***incode*** (for fields, and other initializations less than an int in size), and ***cinit*** (for all other initializations). The size is passed to all these routines, and it is up to the machine dependent routines to ensure that the initializer occupies exactly the right size.

Character strings represent a bit of an exception. If a character string is seen as the initializer for a pointer, the characters making up the string must be put out under a different location counter. When the lexical analyzer sees the quote at the head of a character string, it returns the token STRING, but does not do anything with the contents. The parser calls ***getstr***, which sets up the appropriate location counters and flags, and calls ***lxstr*** to read and process the contents of the

string.

If the string is being used to initialize a character array, *lxstr* calls *putbyte*, which in effect simulates *doinit* for each character read. If the string is used to initialize a character pointer, *lxstr* calls a machine dependent routine, *bycode*, which stashes away each character. The pointer to this string is then returned, and processed normally by *doinit*.

The null at the end of the string is treated as if it were read explicitly by *lxstr*.

Statements

The first pass addresses four main areas; declarations, expressions, initialization, and statements. The statement processing is relatively simple; most of it is carried out in the parser directly. Most of the logic is concerned with allocating label numbers, defining the labels, and branching appropriately. An external symbol, *reached*, is 1 if a statement can be reached, 0 otherwise; this is used to do a bit of simple flow analysis as the program is being parsed, and also to avoid generating the subroutine return sequence if the subroutine cannot "fall through" the last statement.

Conditional branches are handled by generating an expression node, CBRANCH, whose left descendant is the conditional expression and the right descendant is an ICON node containing the internal label number to be branched to. For efficiency, the semantics are that the label is gone to if the condition is *false*.

The switch statement is compiled by collecting the case entries, and an indication as to whether there is a default case; an internal label number is generated for each of these, and remembered in a big array. The expression comprising the value to be switched on is compiled when the switch keyword is encountered, but the expression tree is headed by a special node, FORCE, which tells the code generator to put the expression value into a special distinguished register (this same mechanism is used for processing the return statement). When the end of the switch block is reached, the array containing the case values is sorted, and checked for duplicate entries (an error); if all is correct, the machine dependent routine *genswitch* is called, with this array of labels and values in increasing order. *Genswitch* can assume that the value to be tested is already in the register which is the usual integer return value register.

Optimization

There is a machine independent file, *optim.c*, which contains a relatively short optimization routine, *optim*. Actually the word optimization is something of a misnomer; the results are not optimum, only improved, and the routine is in fact not optional; it must be called for proper operation of the compiler.

Optim is called after an expression tree is built, but before the code generator is called. The essential part of its job is to call *clocal* on the conversion operators. On most machines, the treatment of & is also essential: by this time in the processing, the only node which is a legal descendant of & is NAME. (Possible descendants of * have been eliminated by *buildtree.)* The address of a static name is, almost by definition, a constant, and can be represented by an ICON node on most machines (provided that the loader has enough power). Unfortunately, this is not universally true; on some machine, such as the IBM 370, the issue of addressability rears its ugly head; thus, before turning a NAME node into an ICON node, the machine dependent function *andable* is called.

The optimization attempts of *optim* are currently quite limited. It is primarily concerned with improving the behavior of the compiler with operations one of whose arguments is a constant. In the simplest case, the constant is placed on the right if the operation is commutative. The compiler also makes a limited search for expressions such as

$$(x + a) + b$$

where a and b are constants, and attempts to combine a and b at compile time. A number of special cases are also examined; additions of 0 and multiplications by 1 are removed, although the

correct processing of these cases to get the type of the resulting tree correct is decidedly nontrivial. In some cases, the addition or multiplication must be replaced by a conversion op to keep the types from becoming fouled up. Finally, in cases where a relational operation is being done, and one operand is a constant, the operands are permuted, and the operator altered, if necessary, to put the constant on the right. Finally, multiplications by a power of 2 are changed to shifts.

There are dozens of similar optimizations that can be, and should be, done. It seems likely that this routine will be expanded in the relatively near future.

Machine Dependent Stuff

A number of the first pass machine dependent routines have been discussed above. In general, the routines are short, and easy to adapt from machine to machine. The two exceptions to this general rule are *clocal* and the function prolog and epilog generation routines, *bfcode* and *efcode*.

Clocal has the job of rewriting, if appropriate and desirable, the nodes constructed by *buildtree*. There are two major areas where this is important; NAME nodes and conversion operations. In the case of NAME nodes, *clocal* must rewrite the NAME node to reflect the actual physical location of the name in the machine. In effect, the NAME node must be examined, the symbol table entry found (through the *rval* field of the node), and, based on the storage class of the node, the tree must be rewritten. Automatic variables and parameters are typically rewritten by treating the reference to the variable as a structure reference, off the register which holds the stack or argument pointer; the *stref* routine is set up to be called in this way, and to build the appropriate tree. In the most general case, the tree consists of a unary * node, whose descendant is a + node, with the stack or argument register as left operand, and a constant offset as right operand. In the case of LABEL and internal static nodes, the *rval* field is rewritten to be the negative of the internal label number; a negative *rval* field is taken to be an internal label number. Finally, a name of class REGISTER must be converted into a REG node, and the *rval* field replaced by the register number. In fact, this part of the *clocal* routine is nearly machine independent; only for machines with addressability problems (IBM 370 again!) does it have to be noticeably different,

The conversion operator treatment is rather tricky. It is necessary to handle the application of conversion operators to constants in *clocal*, in order that all constant expressions can have their values known at compile time. In extreme cases, this may mean that some simulation of the arithmetic of the target machine might have to be done in a cross-compiler. In the most common case, conversions from pointer to pointer do nothing. For some machines, however, conversion from byte pointer to short or long pointer might require a shift or rotate operation, which would have to be generated here.

The extension of the portable compiler to machines where the size of a pointer depends on its type would be straightforward, but has not yet been done.

The other major machine dependent issue involves the subroutine prolog and epilog generation. The hard part here is the design of the stack frame and calling sequence; this design issue is discussed elsewhere.[5] The routine *bfcode* is called with the number of arguments the function is defined with, and an array containing the symbol table indices of the declared parameters. *Bfcode* must generate the code to establish the new stack frame, save the return address and previous stack pointer value on the stack, and save whatever registers are to be used for register variables. The stack size and the number of register variables is not known when *bfcode* is called, so these numbers must be referred to by assembler constants, which are defined when they are known (usually in the second pass, after all register variables, automatics, and temporaries have been seen). The final job is to find those parameters which may have been declared register, and generate the code to initialize the register with the value passed on the stack. Once again, for most machines, the general logic of *bfcode* remains the same, but the contents of the *printf* calls in it will change from machine to machine. *efcode* is rather simpler, having just to generate the default return at the end of a function. This may be nontrivial in the case of a function returning a structure or union, however.

There seems to be no really good place to discuss structures and unions, but this is as good a place as any. The C language now supports structure assignment, and the passing of structures as arguments to functions, and the receiving of structures back from functions. This was added rather late to C, and thus to the portable compiler. Consequently, it fits in less well than the older features. Moreover, most of the burden of making these features work is placed on the machine dependent code.

There are both conceptual and practical problems. Conceptually, the compiler is structured around the idea that to compute something, you put it into a register and work on it. This notion causes a bit of trouble on some machines (e.g., machines with 3-address opcodes), but matches many machines quite well. Unfortunately, this notion breaks down with structures. The closest that one can come is to keep the addresses of the structures in registers. The actual code sequences used to move structures vary from the trivial (a multiple byte move) to the horrible (a function call), and are very machine dependent.

The practical problem is more painful. When a function returning a structure is called, this function has to have some place to put the structure value. If it places it on the stack, it has difficulty popping its stack frame. If it places the value in a static temporary, the routine fails to be reentrant. The most logically consistent way of implementing this is for the caller to pass in a pointer to a spot where the called function should put the value before returning. This is relatively straightforward, although a bit tedious, to implement, but means that the caller must have properly declared the function type, even if the value is never used. On some machines, such as the Interdata 8/32, the return value simply overlays the argument region (which on the 8/32 is part of the caller's stack frame). The caller takes care of leaving enough room if the returned value is larger than the arguments. This also assumes that the caller know and declares the function properly.

The PDP-11 and the VAX have stack hardware which is used in function calls and returns; this makes it very inconvenient to use either of the above mechanisms. In these machines, a static area within the called functionis allocated, and the function return value is copied into it on return; the function returns the address of that region. This is simple to implement, but is nonreentrant. However, the function can now be called as a subroutine without being properly declared, without the disaster which would otherwise ensue. No matter what choice is taken, the convention is that the function actually returns the address of the return structure value.

In building expression trees, the portable compiler takes a bit for granted about structures. It assumes that functions returning structures actually return a pointer to the structure, and it assumes that a reference to a structure is actually a reference to its address. The structure assignment operator is rebuilt so that the left operand is the structure being assigned to, but the right operand is the address of the structure being assigned; this makes it easier to deal with

$$a = b = c$$

and similar constructions.

There are four special tree nodes associated with these operations: STASG (structure assignment), STARG (structure argument to a function call), and STCALL and UNARY STCALL (calls of a function with nonzero and zero arguments, respectively). These four nodes are unique in that the size and alignment information, which can be determined by the type for all other objects in C, must be known to carry out these operations; special fields are set aside in these nodes to contain this information, and special intermediate code is used to transmit this information.

First Pass Summary

There are may other issues which have been ignored here, partly to justify the title "tour", and partially because they have seemed to cause little trouble. There are some debugging flags which may be turned on, by giving the compiler's first pass the argument

-X[flags]

Some of the more interesting flags are -Xd for the defining and freeing of symbols, -Xi for

initialization comments, and -Xb for various comments about the building of trees. In many cases, repeating the flag more than once gives more information; thus, -Xddd gives more information than -Xd. In the two pass version of the compiler, the flags should not be set when the output is sent to the second pass, since the debugging output and the intermediate code both go onto the standard output.

We turn now to consideration of the second pass.

Pass Two

Code generation is far less well understood than parsing or lexical analysis, and for this reason the second pass is far harder to discuss in a file by file manner. A great deal of the difficulty is in understanding the issues and the strategies employed to meet them. Any particular function is likely to be reasonably straightforward.

Thus, this part of the paper will concentrate a good deal on the broader aspects of strategy in the code generator, and will not get too intimate with the details.

Overview.

It is difficult to organize a code generator to be flexible enough to generate code for a large number of machines, and still be efficient for any one of them. Flexibility is also important when it comes time to tune the code generator to improve the output code quality. On the other hand, too much flexibility can lead to semantically incorrect code, and potentially a combinatorial explosion in the number of cases to be considered in the compiler.

One goal of the code generator is to have a high degree of correctness. It is very desirable to have the compiler detect its own inability to generate correct code, rather than to produce incorrect code. This goal is achieved by having a simple model of the job to be done (e.g., an expression tree) and a simple model of the machine state (e.g., which registers are free). The act of generating an instruction performs a transformation on the tree and the machine state; hopefully, the tree eventually gets reduced to a single node. If each of these instruction/transformation pairs is correct, and if the machine state model really represents the actual machine, and if the transformations reduce the input tree to the desired single node, then the output code will be correct.

For most real machines, there is no definitive theory of code generation that encompasses all the C operators. Thus the selection of which instruction/transformations to generate, and in what order, will have a heuristic flavor. If, for some expression tree, no transformation applies, or, more seriously, if the heuristics select a sequence of instruction/transformations that do not in fact reduce the tree, the compiler will report its inability to generate code, and abort.

A major part of the code generator is concerned with the model and the transformations, — most of this is machine independent, or depends only on simple tables. The flexibility comes from the heuristics that guide the transformations of the trees, the selection of subgoals, and the ordering of the computation.

The Machine Model

The machine is assumed to have a number of registers, of at most two different types: *A* and *B*. Within each register class, there may be scratch (temporary) registers and dedicated registers (e.g., register variables, the stack pointer, etc.). Requests to allocate and free registers involve only the temporary registers.

Each of the registers in the machine is given a name and a number in the *mac2defs* file; the numbers are used as indices into various tables that describe the registers, so they should be kept small. One such table is the *rstatus* table on file *local2.c*. This table is indexed by register number, and contains expressions made up from manifest constants describing the register types: SAREG for dedicated AREG's, SAREG|STAREG for scratch AREGS's, and SBREG and SBREG|STBREG similarly for BREG's. There are macros that access this information: *isbreg(r)* returns true if register number *r* is a BREG, and *istreg(r)* returns true if register number *r* is a temporary AREG or BREG. Another table, *rnames*, contains the register names; this is used when

putting out assembler code and diagnostics.

The usage of registers is kept track of by an array called *busy*. *Busy[r]* is the number of uses of register *r* in the current tree being processed. The allocation and freeing of registers will be discussed later as part of the code generation algorithm.

General Organization

As mentioned above, the second pass reads lines from the intermediate file, copying through to the output unchanged any lines that begin with a ')', and making note of the information about stack usage and register allocation contained on lines beginning with ']' and '['. The expression trees, whose beginning is indicated by a line beginning with '.', are read and rebuilt into trees. If the compiler is loaded as one pass, the expression trees are immediately available to the code generator.

The actual code generation is done by a hierarchy of routines. The routine *delay* is first given the tree; it attempts to delay some postfix ++ and — computations that might reasonably be done after the smoke clears. It also attempts to handle comma (,) operators by computing the left side expression first, and then rewriting the tree to eliminate the operator. *Delay* calls *codgen* to control the actual code generation process. *Codgen* takes as arguments a pointer to the expression tree, and a second argument that, for socio-historical reasons, is called a *cookie*. The cookie describes a set of goals that would be acceptable for the code generation: these are assigned to individual bits, so they may be logically or'ed together to form a large number of possible goals. Among the possible goals are FOREFF (compute for side effects only; don't worry about the value), INTEMP (compute and store value into a temporary location in memory), INAREG (compute into an A register), INTAREG (compute into a scratch A register), INBREG and INTBREG similarly, FORCC (compute for condition codes), and FORARG (compute it as a function argument; e.g., stack it if appropriate).

Codgen first canonicalizes the tree by calling *canon*. This routine looks for certain transformations that might now be applicable to the tree. One, which is very common and very powerful, is to fold together an indirection operator (UNARY MUL) and a register (REG); in most machines, this combination is addressable directly, and so is similar to a NAME in its behavior. The UNARY MUL and REG are folded together to make another node type called OREG. In fact, in many machines it is possible to directly address not just the cell pointed to by a register, but also cells differing by a constant offset from the cell pointed to by the register. *Canon* also looks for such cases, calling the machine dependent routine *notoff* to decide if the offset is acceptable (for example, in the IBM 370 the offset must be between 0 and 4095 bytes). Another optimization is to replace bit field operations by shifts and masks if the operation involves extracting the field. Finally, a machine dependent routine, *sucomp*, is called that computes the Sethi-Ullman numbers for the tree (see below).

After the tree is canonicalized, *codgen* calls the routine *store* whose job is to select a subtree of the tree to be computed and (usually) stored before beginning the computation of the full tree. *Store* must return a tree that can be computed without need for any temporary storage locations. In effect, the only store operations generated while processing the subtree must be as a response to explicit assignment operators in the tree. This division of the job marks one of the more significant, and successful, departures from most other compilers. It means that the code generator can operate under the assumption that there are enough registers to do its job, without worrying about temporary storage. If a store into a temporary appears in the output, it is always as a direct result of logic in the *store* routine; this makes debugging easier.

One consequence of this organization is that code is not generated by a treewalk. There are theoretical results that support this decision.[7] It may be desirable to compute several subtrees and store them before tackling the whole tree; if a subtree is to be stored, this is known before the code generation for the subtree is begun, and the subtree is computed when all scratch registers are available.

The *store* routine decides what subtrees, if any, should be stored by making use of numbers, called *Sethi-Ullman numbers*, that give, for each subtree of an expression tree, the minimum number of scratch registers required to compile the subtree, without any stores into temporaries.[8] These numbers are computed by the machine-dependent routine *sucomp*, called by *canon*. The basic notion is that, knowing the Sethi-Ullman numbers for the descendants of a node, and knowing the operator of the node and some information about the machine, the Sethi-Ullman number of the node itself can be computed. If the Sethi-Ullman number for a tree exceeds the number of scratch registers available, some subtree must be stored. Unfortunately, the theory behind the Sethi-Ullman numbers applies only to uselessly simple machines and operators. For the rich set of C operators, and for machines with asymmetric registers, register pairs, different kinds of registers, and exceptional forms of addressing, the theory cannot be applied directly. The basic idea of estimation is a good one, however, and well worth applying; the application, especially when the compiler comes to be tuned for high code quality, goes beyond the park of theory into the swamp of heuristics. This topic will be taken up again later, when more of the compiler structure has been described.

After examining the Sethi-Ullman numbers, *store* selects a subtree, if any, to be stored, and returns the subtree and the associated cookie in the external variables *stotree* and *stocook*. If a subtree has been selected, or if the whole tree is ready to be processed, the routine *order* is called, with a tree and cookie. *Order* generates code for trees that do not require temporary locations. *Order* may make recursive calls on itself, and, in some cases, on *codgen*; for example, when processing the operators &&, || and comma (','), that have a left to right evaluation, it is incorrect for *store* examine the right operand for subtrees to be stored. In these cases, *order* will call *codgen* recursively when it is permissible to work on the right operand. A similar issue arises with the ? : operator.

The *order* routine works by matching the current tree with a set of code templates. If a template is discovered that will match the current tree and cookie, the associated assembly language statement or statements are generated. The tree is then rewritten, as specified by the template, to represent the effect of the output instruction(s). If no template match is found, first an attempt is made to find a match with a different cookie; for example, in order to compute an expression with cookie INTEMP (store into a temporary storage location), it is usually necessary to compute the expression into a scratch register first. If all attempts to match the tree fail, the heuristic part of the algorithm becomes dominant. Control is typically given to one of a number of machine-dependent routines that may in turn recursively call *order* to achieve a subgoal of the computation (for example, one of the arguments may be computed into a temporary register). After this subgoal has been achieved, the process begins again with the modified tree. If the machine-dependent heuristics are unable to reduce the tree further, a number of default rewriting rules may be considered appropriate. For example, if the left operand of a + is a scratch register, the + can be replaced by a += operator; the tree may then match a template.

To close this introduction, we will discuss the steps in compiling code for the expression

$$a \mathrel{+}= b$$

where a and b are static variables.

To begin with, the whole expression tree is examined with cookie FOREFF, and no match is found. Search with other cookies is equally fruitless, so an attempt at rewriting is made. Suppose we are dealing with the Interdata 8/32 for the moment. It is recognized that the left hand and right hand sides of the += operator are addressable, and in particular the left hand side has no side effects, so it is permissible to rewrite this as

$$a = a + b$$

and this is done. No match is found on this tree either, so a machine dependent rewrite is done; it is recognized that the left hand side of the assignment is addressable, but the right hand side is not in a register, so *order* is called recursively, being asked to put the right hand side of the assignment into a register. This invocation of *order* searches the tree for a match, and fails. The

machine dependent rule for + notices that the right hand operand is addressable; it decides to put the left operand into a scratch register. Another recursive call to *order* is made, with the tree consisting solely of the leaf *a*, and the cookie asking that the value be placed into a scratch register. This now matches a template, and a load instruction is emitted. The node consisting of *a* is rewritten in place to represent the register into which *a* is loaded, and this third call to *order* returns. The second call to *order* now finds that it has the tree

reg + *b*

to consider. Once again, there is no match, but the default rewriting rule rewrites the + as a += operator, since the left operand is a scratch register. When this is done, there is a match: in fact,

reg += *b*

simply describes the effect of the add instruction on a typical machine. After the add is emitted, the tree is rewritten to consist merely of the register node, since the result of the add is now in the register. This agrees with the cookie passed to the second invocation of *order*, so this invocation terminates, returning to the first level. The original tree has now become

a = **reg**

which matches a template for the store instruction. The store is output, and the tree rewritten to become just a single register node. At this point, since the top level call to *order* was interested only in side effects, the call to *order* returns, and the code generation is completed; we have generated a load, add, and store, as might have been expected.

The effect of machine architecture on this is considerable. For example, on the Honeywell 6000, the machine dependent heuristics recognize that there is an "add to storage" instruction, so the strategy is quite different; *b* is loaded in to a register, and then an add to storage instruction generated to add this register in to *a*. The transformations, involving as they do the semantics of C, are largely machine independent. The decisions as to when to use them, however, are almost totally machine dependent.

Having given a broad outline of the code generation process, we shall next consider the heart of it: the templates. This leads naturally into discussions of template matching and register allocation, and finally a discussion of the machine dependent interfaces and strategies.

The Templates

The templates describe the effect of the target machine instructions on the model of computation around which the compiler is organized. In effect, each template has five logical sections, and represents an assertion of the form:

> **If** we have a subtree of a given shape (1), and we have a goal (cookie) or goals to achieve (2), and we have sufficient free resources (3), **then** we may emit an instruction or instructions (4), and rewrite the subtree in a particular manner (5), and the rewritten tree will achieve the desired goals.

These five sections will be discussed in more detail later. First, we give an example of a template:

```
ASG PLUS,    INAREG,
             SAREG,      TINT,
             SNAME,      TINT,
                         0,          RLEFT,
                         "           add         AL,AR\n",
```

The top line specifies the operator (+=) and the cookie (compute the value of the subtree into an AREG). The second and third lines specify the left and right descendants, respectively, of the += operator. The left descendant must be a REG node, representing an A register, and have integer type, while the right side must be a NAME node, and also have integer type. The fourth line contains the resource requirements (no scratch registers or temporaries needed), and the rewriting rule

(replace the subtree by the left descendant). Finally, the quoted string on the last line represents the output to the assembler: lower case letters, tabs, spaces, etc. are copied *verbatim*. to the output; upper case letters trigger various macro-like expansions. Thus, **AL** would expand into the **A**ddress form of the **L**eft operand — presumably the register number. Similarly, **AR** would expand into the name of the right operand. The *add* instruction of the last section might well be emitted by this template.

In principle, it would be possible to make separate templates for all legal combinations of operators, cookies, types, and shapes. In practice, the number of combinations is very large. Thus, a considerable amount of mechanism is present to permit a large number of subtrees to be matched by a single template. Most of the shape and type specifiers are individual bits, and can be logically or'ed together. There are a number of special descriptors for matching classes of operators. The cookies can also be combined. As an example of the kind of template that really arises in practice, the actual template for the Interdata 8/32 that subsumes the above example is:

```
ASG OPSIMP,  INAREG|FORCC,
             SAREG,        TINT|TUNSIGNED|TPOINT,
             SAREG|SNAME|SOREG|SCON, TINT|TUNSIGNED|TPOINT,
                           0,              RLEFT|RESCC,
                           "               OI          AL,AR\n",
```

Here, OPSIMP represents the operators +, −, |, &, and ^. The **OI** macro in the output string expands into the appropriate **I**nteger **O**pcode for the operator. The left and right sides can be integers, unsigned, or pointer types. The right side can be, in addition to a name, a register, a memory location whose address is given by a register and displacement (OREG), or a constant. Finally, these instructions set the condition codes, and so can be used in condition contexts: the cookie and rewriting rules reflect this.

The Template Matching Algorithm.

The heart of the second pass is the template matching algorithm, in the routine *match*. *Match* is called with a tree and a cookie; it attempts to match the given tree against some template that will transform it according to one of the goals given in the cookie. If a match is successful, the transformation is applied; *expand* is called to generate the assembly code, and then *reclaim* rewrites the tree, and reclaims the resources, such as registers, that might have become free as a result of the generated code.

This part of the compiler is among the most time critical. There is a spectrum of implementation techniques available for doing this matching. The most naive algorithm simply looks at the templates one by one. This can be considerably improved upon by restricting the search for an acceptable template. It would be possible to do better than this if the templates were given to a separate program that ate them and generated a template matching subroutine. This would make maintenance of the compiler much more complicated, however, so this has not been done.

The matching algorithm is actually carried out by restricting the range in the table that must be searched for each opcode. This introduces a number of complications, however, and needs a bit of sympathetic help by the person constructing the compiler in order to obtain best results. The exact tuning of this algorithm continues; it is best to consult the code and comments in *match* for the latest version.

In order to match a template to a tree, it is necessary to match not only the cookie and the op of the root, but also the types and shapes of the left and right descendants (if any) of the tree. A convention is established here that is carried out throughout the second pass of the compiler. If a node represents a unary operator, the single descendant is always the "left" descendant. If a node represents a unary operator or a leaf node (no descendants) the "right" descendant is taken by convention to be the node itself. This enables templates to easily match leaves and conversion operators, for example, without any additional mechanism in the matching program.

The type matching is straightforward; it is possible to specify any combination of basic types, general pointers, and pointers to one or more of the basic types. The shape matching is

somewhat more complicated, but still pretty simple. Templates have a collection of possible operand shapes on which the opcode might match. In the simplest case, an *add* operation might be able to add to either a register variable or a scratch register, and might be able (with appropriate help from the assembler) to add an integer constant (ICON), a static memory cell (NAME), or a stack location (OREG).

It is usually attractive to specify a number of such shapes, and distinguish between them when the assembler output is produced. It is possible to describe the union of many elementary shapes such as ICON, NAME, OREG, AREG or BREG (both scratch and register forms), etc. To handle at least the simple forms of indirection, one can also match some more complicated forms of trees; STARNM and STARREG can match more complicated trees headed by an indirection operator, and SFLD can match certain trees headed by a FLD operator: these patterns call machine dependent routines that match the patterns of interest on a given machine. The shape SWADD may be used to recognize NAME or OREG nodes that lie on word boundaries: this may be of some importance on word-addressed machines. Finally, there are some special shapes: these may not be used in conjunction with the other shapes, but may be defined and extended in machine dependent ways. The special shapes SZERO, SONE, and SMONE are predefined and match constants 0, 1, and -1, respectively; others are easy to add and match by using the machine dependent routine *special*.

When a template has been found that matches the root of the tree, the cookie, and the shapes and types of the descendants, there is still one bar to a total match: the template may call for some resources (for example, a scratch register). The routine *allo* is called, and it attempts to allocate the resources. If it cannot, the match fails; no resources are allocated. If successful, the allocated resources are given numbers 1, 2, etc. for later reference when the assembly code is generated. The routines *expand* and *reclaim* are then called. The *match* routine then returns a special value, MDONE. If no match was found, the value MNOPE is returned; this is a signal to the caller to try more cookie values, or attempt a rewriting rule. *Match* is also used to select rewriting rules, although the way of doing this is pretty straightforward. A special cookie, FORREW, is used to ask *match* to search for a rewriting rule. The rewriting rules are keyed to various opcodes; most are carried out in *order*. Since the question of when to rewrite is one of the key issues in code generation, it will be taken up again later.

Register Allocation.

The register allocation routines, and the allocation strategy, play a central role in the correctness of the code generation algorithm. If there are bugs in the Sethi-Ullman computation that cause the number of needed registers to be underestimated, the compiler may run out of scratch registers; it is essential that the allocator keep track of those registers that are free and busy, in order to detect such conditions.

Allocation of registers takes place as the result of a template match; the routine *allo* is called with a word describing the number of A registers, B registers, and temporary locations needed. The allocation of temporary locations on the stack is relatively straightforward, and will not be further covered; the bookkeeping is a bit tricky, but conceptually trivial, and requests for temporary space on the stack will never fail.

Register allocation is less straightforward. The two major complications are *pairing* and *sharing*. In many machines, some operations (such as multiplication and division), and/or some types (such as longs or double precision) require even/odd pairs of registers. Operations of the first type are exceptionally difficult to deal with in the compiler; in fact, their theoretical properties are rather bad as well.[9] The second issue is dealt with rather more successfully; a machine dependent function called *szty(t)* is called that returns 1 or 2, depending on the number of A registers required to hold an object of type t. If *szty* returns 2, an even/odd pair of A registers is allocated for each request.

The other issue, sharing, is more subtle, but important for good code quality. When registers are allocated, it is possible to reuse registers that hold address information, and use them to contain the values computed or accessed. For example, on the IBM 360, if register 2 has a pointer to

an integer in it, we may load the integer into register 2 itself by saying:

```
L    2,0(2)
```

If register 2 had a byte pointer, however, the sequence for loading a character involves clearing the target register first, and then inserting the desired character:

```
SR   3,3
IC   3,0(2)
```

In the first case, if register 3 were used as the target, it would lead to a larger number of registers used for the expression than were required; the compiler would generate inefficient code. On the other hand, if register 2 were used as the target in the second case, the code would simply be wrong. In the first case, register 2 can be *shared* while in the second, it cannot.

In the specification of the register needs in the templates, it is possible to indicate whether required scratch registers may be shared with possible registers on the left or the right of the input tree. In order that a register be shared, it must be scratch, and it must be used only once, on the appropriate side of the tree being compiled.

The *allo* routine thus has a bit more to do than meets the eye; it calls *freereg* to obtain a free register for each A and B register request. *Freereg* makes multiple calls on the routine *usable* to decide if a given register can be used to satisfy a given need. *Usable* calls *shareit* if the register is busy, but might be shared. Finally, *shareit* calls *ushare* to decide if the desired register is actually in the appropriate subtree, and can be shared.

Just to add additional complexity, on some machines (such as the IBM 370) it is possible to have "double indexing" forms of addressing; these are represented by OREGS's with the base and index registers encoded into the register field. While the register allocation and deallocation *per se* is not made more difficult by this phenomenon, the code itself is somewhat more complex.

Having allocated the registers and expanded the assembly language, it is time to reclaim the resources; the routine *reclaim* does this. Many operations produce more than one result. For example, many arithmetic operations may produce a value in a register, and also set the condition codes. Assignment operations may leave results both in a register and in memory. *Reclaim* is passed three parameters; the tree and cookie that were matched, and the rewriting field of the template. The rewriting field allows the specification of possible results; the tree is rewritten to reflect the results of the operation. If the tree was computed for side effects only (FOREFF), the tree is freed, and all resources in it reclaimed. If the tree was computed for condition codes, the resources are also freed, and the tree replaced by a special node type, FORCC. Otherwise, the value may be found in the left argument of the root, the right argument of the root, or one of the temporary resources allocated. In these cases, first the resources of the tree, and the newly allocated resources, are freed; then the resources needed by the result are made busy again. The final result must always match the shape of the input cookie; otherwise, the compiler error "cannot reclaim" is generated. There are some machine dependent ways of preferring results in registers or memory when there are multiple results matching multiple goals in the cookie.

The Machine Dependent Interface

The files *order.c*, *local2.c*, and *table.c*, as well as the header file *mac2defs*, represent the machine dependent portion of the second pass. The machine dependent portion can be roughly divided into two: the easy portion and the hard portion. The easy portion tells the compiler the names of the registers, and arranges that the compiler generate the proper assembler formats, opcode names, location counters, etc. The hard portion involves the Sethi–Ullman computation, the rewriting rules, and, to some extent, the templates. It is hard because there are no real algorithms that apply; most of this portion is based on heuristics. This section discusses the easy portion; the next several sections will discuss the hard portion.

If the compiler is adapted from a compiler for a machine of similar architecture, the easy part is indeed easy. In *mac2defs*, the register numbers are defined, as well as various parameters for the stack frame, and various macros that describe the machine architecture. If double indexing

is to be permitted, for example, the symbol R2REGS is defined. Also, a number of macros that are involved in function call processing, especially for unusual function call mechanisms, are defined here.

In *local2.c*, a large number of simple functions are defined. These do things such as write out opcodes, register names, and address forms for the assembler. Part of the function call code is defined here; that is nontrivial to design, but typically rather straightforward to implement. Among the easy routines in *order.c* are routines for generating a created label, defining a label, and generating the arguments of a function call.

These routines tend to have a local effect, and depend on a fairly straightforward way on the target assembler and the design decisions already made about the compiler. Thus they will not be further treated here.

The Rewriting Rules

When a tree fails to match any template, it becomes a candidate for rewriting. Before the tree is rewritten, the machine dependent routine *nextcook* is called with the tree and the cookie; it suggests another cookie that might be a better candidate for the matching of the tree. If all else fails, the templates are searched with the cookie FORREW, to look for a rewriting rule. The rewriting rules are of two kinds; for most of the common operators, there are machine dependent rewriting rules that may be applied; these are handled by machine dependent functions that are called and given the tree to be computed. These routines may recursively call *order* or *codgen* to cause certain subgoals to be achieved; if they actually call for some alteration of the tree, they return 1, and the code generation algorithm recanonicalizes and tries again. If these routines choose not to deal with the tree, the default rewriting rules are applied.

The assignment ops, when rewritten, call the routine *setasg*. This is assumed to rewrite the tree at least to the point where there are no side effects in the left hand side. If there is still no template match, a default rewriting is done that causes an expression such as

$$a \mathrel{+}= b$$

to be rewritten as

$$a = a + b$$

This is a useful default for certain mixtures of strange types (for example, when a is a bit field and b an character) that otherwise might need separate table entries.

Simple assignment, structure assignment, and all forms of calls are handled completely by the machine dependent routines. For historical reasons, the routines generating the calls return 1 on failure, 0 on success, unlike the other routines.

The machine dependent routine *setbin* handles binary operators; it too must do most of the job. In particular, when it returns 0, it must do so with the left hand side in a temporary register. The default rewriting rule in this case is to convert the binary operator into the associated assignment operator; since the left hand side is assumed to be a temporary register, this preserves the semantics and often allows a considerable saving in the template table.

The increment and decrement operators may be dealt with with the machine dependent routine *setincr*. If this routine chooses not to deal with the tree, the rewriting rule replaces

$$x \mathrel{+}+$$

by

$$(\,(x \mathrel{+}= 1) - 1)$$

which preserves the semantics. Once again, this is not too attractive for the most common cases, but can generate close to optimal code when the type of x is unusual.

Finally, the indirection (UNARY MUL) operator is also handled in a special way. The machine dependent routine *offstar* is extremely important for the efficient generation of code.

Offstar is called with a tree that is the direct descendant of a UNARY MUL node; its job is to transform this tree so that the combination of UNARY MUL with the transformed tree becomes addressable. On most machines, *offstar* can simply compute the tree into an A or B register, depending on the architecture, and then *canon* will make the resulting tree into an OREG. On many machines, *offstar* can profitably choose to do less work than computing its entire argument into a register. For example, if the target machine supports OREGS with a constant offset from a register, and *offstar* is called with a tree of the form

expr + const

where *const* is a constant, then *offstar* need only compute *expr* into the appropriate form of register. On machines that support double indexing, *offstar* may have even more choice as to how to proceed. The proper tuning of *offstar*, which is not typically too difficult, should be one of the first tries at optimization attempted by the compiler writer.

The Sethi-Ullman Computation

The heart of the heuristics is the computation of the Sethi-Ullman numbers. This computation is closely linked with the rewriting rules and the templates. As mentioned before, the Sethi-Ullman numbers are expected to estimate the number of scratch registers needed to compute the subtrees without using any stores. However, the original theory does not apply to real machines. For one thing, the theory assumes that all registers are interchangeable. Real machines have general purpose, floating point, and index registers, register pairs, etc. The theory also does not account for side effects; this rules out various forms of pathology that arise from assignment and assignment ops. Condition codes are also undreamed of. Finally, the influence of types, conversions, and the various addressability restrictions and extensions of real machines are also ignored.

Nevertheless, for a "useless" theory, the basic insight of Sethi and Ullman is amazingly useful in a real compiler. The notion that one should attempt to estimate the resource needs of trees before starting the code generation provides a natural means of splitting the code generation problem, and provides a bit of redundancy and self checking in the compiler. Moreover, if writing the Sethi-Ullman routines is hard, describing, writing, and debugging the alternative (routines that attempt to free up registers by stores into temporaries "on the fly") is even worse. Nevertheless, it should be clearly understood that these routines exist in a realm where there is no "right" way to write them; it is an art, the realm of heuristics, and, consequently, a major source of bugs in the compiler. Often, the early, crude versions of these routines give little trouble; only after the compiler is actually working and the code quality is being improved do serious problem have to be faced. Having a simple, regular machine architecture is worth quite a lot at this time.

The major problems arise from asymmetries in the registers: register pairs, having different kinds of registers, and the related problem of needing more than one register (frequently a pair) to store certain data types (such as longs or doubles). There appears to be no general way of treating this problem; solutions have to be fudged for each machine where the problem arises. On the Honeywell 66, for example, there are only two general purpose registers, so a need for a pair is the same as the need for two registers. On the IBM 370, the register pair (0,1) is used to do multiplications and divisions; registers 0 and 1 are not generally considered part of the scratch registers, and so do not require allocation explicitly. On the Interdata 8/32, after much consideration, the decision was made not to try to deal with the register pair issue; operations such as multiplication and division that required pairs were simply assumed to take all of the scratch registers. Several weeks of effort had failed to produce an algorithm that seemed to have much chance of running successfully without inordinate debugging effort. The difficulty of this issue should not be minimized; it represents one of the main intellectual efforts in porting the compiler. Nevertheless, this problem has been fudged with a degree of success on nearly a dozen machines, so the compiler writer should not abandon hope.

The Sethi-Ullman computations interact with the rest of the compiler in a number of rather subtle ways. As already discussed, the *store* routine uses the Sethi-Ullman numbers to decide which subtrees are too difficult to compute in registers, and must be stored. There are also subtle

interactions between the rewriting routines and the Sethi-Ullman numbers. Suppose we have a tree such as

A - B

where *A* and *B* are expressions; suppose further that *B* takes two registers, and *A* one. It is possible to compute the full expression in two registers by first computing *B*, and then, using the scratch register used by *B*, but not containing the answer, compute *A*. The subtraction can then be done, computing the expression. (Note that this assumes a number of things, not the least of which are register-to-register subtraction operators and symmetric registers.) If the machine dependent routine ***setbin***, however, is not prepared to recognize this case and compute the more difficult side of the expression first, the Sethi-Ullman number must be set to three. Thus, the Sethi-Ullman number for a tree should represent the code that the machine dependent routines are actually willing to generate.

The interaction can go the other way. If we take an expression such as

** (p + i)*

where *p* is a pointer and *i* an integer, this can probably be done in one register on most machines. Thus, its Sethi-Ullman number would probably be set to one. If double indexing is possible in the machine, a possible way of computing the expression is to load both *p* and *i* into registers, and then use double indexing. This would use two scratch registers; in such a case, it is possible that the scratch registers might be unobtainable, or might make some other part of the computation run out of registers. The usual solution is to cause ***offstar*** to ignore opportunities for double indexing that would tie up more scratch registers than the Sethi-Ullman number had reserved.

In summary, the Sethi-Ullman computation represents much of the craftsmanship and artistry in any application of the portable compiler. It is also a frequent source of bugs. Algorithms are available that will produce nearly optimal code for specialized machines, but unfortunately most existing machines are far removed from these ideals. The best way of proceeding in practice is to start with a compiler for a similar machine to the target, and proceed very carefully.

Register Allocation

After the Sethi-Ullman numbers are computed, ***order*** calls a routine, ***rallo***, that does register allocation, if appropriate. This routine does relatively little, in general; this is especially true if the target machine is fairly regular. There are a few cases where it is assumed that the result of a computation takes place in a particular register; switch and function return are the two major places. The expression tree has a field, ***rall***, that may be filled with a register number; this is taken to be a preferred register, and the first temporary register allocated by a template match will be this preferred one, if it is free. If not, no particular action is taken; this is just a heuristic. If no register preference is present, the field contains NOPREF. In some cases, the result must be placed in a given register, no matter what. The register number is placed in ***rall***, and the mask MUSTDO is logically or'ed in with it. In this case, if the subtree is requested in a register, and comes back in a register other than the demanded one, it is moved by calling the routine ***rmove***. If the target register for this move is busy, it is a compiler error.

Note that this mechanism is the only one that will ever cause a register-to-register move between scratch registers (unless such a move is buried in the depths of some template). This simplifies debugging. In some cases, there is a rather strange interaction between the register allocation and the Sethi-Ullman number; if there is an operator or situation requiring a particular register, the allocator and the Sethi-Ullman computation must conspire to ensure that the target register is not being used by some intermediate result of some far-removed computation. This is most easily done by making the special operation take all of the free registers, preventing any other partially-computed results from cluttering up the works.

Compiler Bugs

The portable compiler has an excellent record of generating correct code. The requirement for reasonable cooperation between the register allocation, Sethi-Ullman computation, rewriting rules, and templates builds quite a bit of redundancy into the compiling process. The effect of this is that, in a surprisingly short time, the compiler will start generating correct code for those programs that it can compile. The hard part of the job then becomes finding and eliminating those situations where the compiler refuses to compile a program because it knows it cannot do it right. For example, a template may simply be missing; this may either give a compiler error of the form "no match for op ..." , or cause the compiler to go into an infinite loop applying various rewriting rules. The compiler has a variable, *nrecur*, that is set to 0 at the beginning of an expressions, and incremented at key spots in the compilation process; if this parameter gets too large, the compiler decides that it is in a loop, and aborts. Loops are also characteristic of botches in the machine-dependent rewriting rules. Bad Sethi-Ullman computations usually cause the scratch registers to run out; this often means that the Sethi-Ullman number was underestimated, so *store* did not store something it should have; alternatively, it can mean that the rewriting rules were not smart enough to find the sequence that *sucomp* assumed would be used.

The best approach when a compiler error is detected involves several stages. First, try to get a small example program that steps on the bug. Second, turn on various debugging flags in the code generator, and follow the tree through the process of being matched and rewritten. Some flags of interest are –e, which prints the expression tree, –r, which gives information about the allocation of registers, –a, which gives information about the performance of *rallo* , and –o, which gives information about the behavior of *order*. This technique should allow most bugs to be found relatively quickly.

Unfortunately, finding the bug is usually not enough; it must also be fixed! The difficulty arises because a fix to the particular bug of interest tends to break other code that already works. Regression tests, tests that compare the performance of a new compiler against the performance of an older one, are very valuable in preventing major catastrophes.

Summary and Conclusion

The portable compiler has been a useful tool for providing C capability on a large number of diverse machines, and for testing a number of theoretical constructs in a practical setting. It has many blemishes, both in style and functionality. It has been applied to many more machines than first anticipated, of a much wider range than originally dreamed of. Its use has also spread much faster than expected, leaving parts of the compiler still somewhat raw in shape.

On the theoretical side, there is some hope that the skeleton of the *sucomp* routine could be generated for many machines directly from the templates; this would give a considerable boost to the portability and correctness of the compiler, but might affect tunability and code quality. There is also room for more optimization, both within *optim* and in the form of a portable "peephole" optimizer.

On the practical, development side, the compiler could probably be sped up and made smaller without doing too much violence to its basic structure. Parts of the compiler deserve to be rewritten; the initialization code, register allocation, and parser are prime candidates. It might be that doing some or all of the parsing with a recursive descent parser might save enough space and time to be worthwhile; it would certainly ease the problem of moving the compiler to an environment where *Yacc* is not already present.

Finally, I would like to thank the many people who have sympathetically, and even enthusiastically, helped me grapple with what has been a frustrating program to write, test, and install. D. M. Ritchie and E. N. Pinson provided needed early encouragement and philosophical guidance; M. E. Lesk, R. Muha, T. G. Peterson, G. Riddle, L. Rosler, R. W. Mitze, B. R. Rowland, S. I. Feldman, and T. B. London have all contributed ideas, gripes, and all, at one time or another, climbed "into the pits" with me to help debug. Without their help this effort would have not been possible; with it, it was often kind of fun.

References

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
2. S. C. Johnson, "Lint, a C Program Checker," Comp. Sci. Tech. Rep. No. 65, 1978. updated version TM 78-1273-3
3. A. Snyder, *A Portable Compiler for the Language C,* Master's Thesis, M.I.T., Cambridge, Mass., 1974.
4. S. C. Johnson, "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages,* pp. 97-104, January 1978.
5. M. E. Lesk, S. C. Johnson, and D. M. Ritchie, *The C Language Calling Sequence,* 1977.
6. S. C. Johnson, "Yacc — Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey, July 1975.
7. A. V. Aho and S. C. Johnson, "Optimal Code Generation for Expression Trees," *J. Assoc. Comp. Mach.*, vol. 23, no. 3, pp. 488-501, 1975. Also in *Proc. ACM Symp. on Theory of Computing,* pp. 207-217, 1975.
8. R. Sethi and J. D. Ullman, "The Generation of Optimal Code for Arithmetic Expressions," *J. Assoc. Comp. Mach.*, vol. 17, no. 4, pp. 715-728, October 1970. Reprinted as pp. 229-247 in *Compiler Techniques,* ed. B. W. Pollack, Auerbach, Princeton NJ (1972).
9. A. V. Aho, S. C. Johnson, and J. D. Ullman, "Code Generation for Machines with Multiregister Operations," *Proc. 4th ACM Symp. on Principles of Programming Languages,* pp. 21-28, January 1977.

A Portable Fortran 77 Compiler

S. I. Feldman

P. J. Weinberger

Bell Laboratories
Murray Hill, New Jersey 07974

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.

1 August 1978

Berkeley Notes

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).

June 1983

† UNIX is a trademark of Bell Laboratories.

Table of Contents

A Portable Fortran 77 Compiler

S. I. Feldman

P. J. Weinberger

Bell Laboratories
Murray Hill, New Jersey 07974

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**.
−g	Have the compiler produce additional symbol table information for *dbx(1)*. This only applies on the Vax UNIX system. Do not use with **−O**.
−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	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.
−w	Suppress all warning messages.
−w66	Suppress warnings about Fortran 66 features used.
−u	Make the default type of a variable **undefined** (see section 2.3).
−C	Compile code that checks that subscripts are within array bounds.
−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 and and EFL source programs are pre-processed into Fortran files, but those files are not compiled or removed.
−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).
−O	Invoke the object code optimizer. Do not use 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.

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 variables. A double complex version of every **complex** built-in function is provided. The specific function names begin with **z** instead of **c**.

2.2. Internal Files

The Fortran 77 standard introduces "internal files" (memory arrays), 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 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.

2.5. Automatic Storage

Two new keywords are recognized, **static** and **automatic.** These keywords may appear as "types" in type statements and in **implicit** statements. Local variables are static by default; there is exactly one copy of the datum, and its value is retained between calls. There is one copy of each variable declared **automatic** for each invocation of the procedure. Automatic variables may not appear in **equivalence, data,** or **save** statements.

2.6. Source Input Format

The Standard expects input to the compiler to be in 72-column format: except in comment lines, the first five characters are the statement number, the next is the continuation character, and the next 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. Regardless of the setting of the flag, keywords will only be recognized in lower case.

2.7. Include Statement

The statement

```
include 'stuff'
```

is replaced by the contents of the file **stuff**; **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** compiler flag causes non-standard loops to be generated.

2.13. Commas in Formatted Input

The I/O system attempts to be more lenient than the Standard when it seems worthwhile. When doing a formatted read of non-character variables, commas may be used as value separators in the input record, overriding the field lengths given in the format statement. Thus, the format

```
(i10, f20.10, i4)
```

will read the record

```
−345,.05e−3,12
```

correctly.

2.14. Short Integers

On machines that support halfword integers, the compiler accepts declarations of type **integer*2.** (Ordinary integers follow the Fortran rules about occupying the same space as a REAL variable; they are assumed to be of C type **long int**; halfword integers are of C type **short int**.) An expression involving only objects of type **integer*2** is of that type. Generic functions return short or long integers depending on the actual types of their arguments. If a procedure is compiled using the **−i2** flag, all small integer constants will be of type **integer*2.** If the precision of an integer-valued intrinsic function is not determined by the generic function rules, one will be chosen that returns the prevailing length (**integer*2** when the **−i2** command flag is in effect). When the **−i2** option is in effect, all quantities of type **logical** will be short. Note that these short integer and logical quantities do not obey the standard rules for storage association.

2.15. Additional Intrinsic Functions

This compiler supports all of the intrinsic functions specified in the Fortran 77 Standard. In addition, there are functions for performing bitwise Boolean operations (**or**, **and**, **xor**, and **not**) and for accessing the UNIX command arguments (**getarg** and **iargc**) and environment (**getenv**).

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 no carriage control is recognized on formatted output, except by special arrangement [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 library procedure names have embedded underscores to avoid clashes with user-assigned subroutine names.

4.2. Data Representations

The following is a table of corresponding Fortran and C declarations:

Fortran	**C**
integer*2 x	short int x;
integer x	long int x;
logical x	long int x;
real x	float x;
double precision x	double x;
complex x	struct { float r, i; } x;
double complex x	struct { double dr, di; } x;
character*6 x	char x[6];

(By the rules of Fortran, **integer, logical,** and **real** data occupy the same amount of memory.)

4.3. Return Values

A function of type **integer, logical, real,** or **double precision** declared as a C function returns the corresponding type. A **complex** or **double complex** function is equivalent to a C routine with an additional initial argument that points to the place where the return value is to be stored. Thus,

```
complex function f( . . . )
```

is equivalent to

```
f_(temp, . . .)
struct { float r, i; } *temp;
 . . .
```

A character-valued function is equivalent to a C routine with two extra initial arguments: a data address and a length. Thus,

```
character*15 function g( . . . )
```

is equivalent to

```
g_(result, length, . . .)
char result[ ];
long int length;
 . . .
```

and could be invoked in C by

```
char chars[15];
 . . .
g_(chars, 15L, . . . );
```

Subroutines are invoked as if they were **integer**-valued functions whose value specifies which alternate return to use. Alternate return arguments (statement labels) are not passed to the function, but are used to do an indexed branch in the calling procedure. (If the subroutine has no entry points with alternate return arguments, the returned value is undefined.) The statement

```
call nret(*1, *2, *3)
```

is treated exactly as if it were the computed **goto**

```
goto (1, 2, 3), nret( )
```

4.4. Argument Lists

All Fortran arguments are passed by address. In addition, for every argument that is of type character or that is a dummy procedure, an argument giving the length of the value is passed. (The string lengths are **long int** quantities passed by value.) The order of arguments is then:

Extra arguments for complex and character functions
Address for each datum or function
A **long int** for each character or procedure argument

Thus, the call in

```
external f
character*7 s
integer b(3)
 . . .
call sam(f, b(2), s)
```

is equivalent to that in

```
int f();
char s[7];
long int b[3];
 . . .
sam_(f, &b[1], s, 0L, 7L);
```

Note that the first element of a C array always has subscript zero, but Fortran arrays begin at 1 by default. Fortran arrays are stored in column-major order, C arrays are stored in row-major order.

5. FILE FORMATS

5.1. Structure of Fortran Files

Fortran requires four kinds of external files: sequential formatted and unformatted, and direct formatted and unformatted. On UNIX systems, these are all implemented as ordinary files which are assumed to have the proper internal structure.

Fortran I/O is based on *records*. When a direct file is opened in a Fortran program, the record length of the records must be given, and this is used by the Fortran I/O system to make the file look as if it is made up of records of the given length. In the special case that the record length is given as 1, the files are not considered to be divided into records, but are treated as byte-addressable byte strings; that is, as ordinary UNIX file system files. (A read or write request on such a file keeps consuming bytes until satisfied, rather than being

restricted to a single record.)

The peculiar requirements on sequential unformatted files make it unlikely that they will ever be read or written by any means except Fortran I/O statements. Each record is preceded and followed by an integer containing the record's length in bytes.

The Fortran I/O system breaks sequential formatted files into records while reading by using each newline as a record separator. The result of reading off the end of a record is undefined according to the Standard. The I/O system is permissive and treats the record as being extended by blanks. On output, the I/O system will write a newline at the end of each record. It is also possible for programs to write newlines for themselves. This is an error, but the only effect will be that the single record the user thought he wrote will be treated as more than one record when being read or backspaced over.

5.2. Portability Considerations

The Fortran I/O system uses only the facilities of the standard C I/O library, a widely available and fairly portable package, with the following two nonstandard features: the I/O system needs to know whether a file can be used for direct I/O, and whether or not it is possible to backspace. Both of these facilities are implemented using the **fseek** routine, so there is a routine **canseek** which determines if **fseek** will have the desired effect. Also, the **inquire** statement provides the user with the ability to find out if two files are the same, and to get the name of an already opened file in a form which would enable the program to reopen it. Therefore there are two routines which depend on facilities of the operating system to provide these two services. In any case, the I/O system runs on the PDP-11, VAX-11/780, and Interdata 8/32 UNIX systems.

5.3. Pre-Connected Files and File Positions

Units 5, 6, and 0 are preconnected when the program starts. Unit 5 is connected to the standard input, unit 6 is connected to the standard output, and unit 0 is connected to the standard error unit. All are connected for sequential formatted I/O.

All the other units are also preconnected when execution begins. Unit *n* is connected to a file named **fort.***n*. These files need not exist, nor will they be created unless their units are used without first executing an **open.** The default connection is for sequential formatted I/O.

The Standard does not specify where a file which has been explicitly **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 'fseek' function. 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

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

```
parameter (x=17, y=x/3, pi=3.14159d0, s='hello')
```

The type of each parameter name is governed by the same implicit and explicit rules as for a variable. The right side of each equal sign must be a constant expression (an expression made up of constants, operators, and already defined parameters).

3.4. Array Declarations

Arrays may now have as many as seven dimensions. (Only three were permitted in 1966.) The lower bound of each dimension may be declared to be other than 1 by using a colon. Furthermore, an adjustable array bound may be an integer expression involving constants, arguments, and variables in **common.**

```
real a(-5:3, 7, m:n), b(n+1:2*n)
```

The upper bound on the last dimension of an array argument may be denoted by an asterisk to indicate that the upper bound is not specified:

```
integer a(5, *),  b(*), c(0:1, -2:*)
```

3.5. SAVE Statement

A poorly known rule of Fortran 66 is that local variables in a procedure do not necessarily retain their values between invocations of that procedure. At any instant in the execution of a program, if a common block is declared neither in the currently executing procedure nor in any of the procedures in the chain of callers, all of the variables in that common block also become undefined. (The only exceptions are variables that have been defined in a **data** statement and never changed.) These rules permit overlay and stack implementations for the affected variables. Fortran 77 permits one to specify that certain variables and common blocks are to retain their values between invocations. The declaration

```
save a, /b/, c
```

leaves the values of the variables **a** and **c** and all of the contents of common block **b** unaffected by a return. The simple declaration

```
save
```

has this effect on all variables and common blocks in the procedure. A common block must be **saved** in every procedure in which it is declared if the desired effect is to occur.

3.6. INTRINSIC Statement

All of the functions specified in the Standard are in a single category, "intrinsic functions", rather than being divided into "intrinsic" and "basic external" functions. If an intrinsic function is to be passed to another procedure, it must be declared **intrinsic.** Declaring it **external** (as in Fortran 66) causes a function other than the built-in one to be passed.

4. Expressions

4.1. Character Constants

Character string constants are marked by strings surrounded by apostrophes. If an apostrophe is to be included in a constant, it is repeated:

```
'abc'
'ain''t'
```

There are no null (zero-length) character strings in Fortran 77. Our compiler has two

different quotation marks, " ' " and " " ". (See section 2.9 in the main text.)

4.2. Concatenation

One new operator has been added, character string concatenation, marked by a double slash "//". The result of a concatenation is the string containing the characters of the left operand followed by the characters of the right operand. The strings

```
'ab' // 'cd'
'abcd'
```

are equal. The strings being concatenated must be of constant length in all concatenations that are not the right sides of assignments. (The only concatenation expressions in which a character string declared adjustable with a "*(*)" modifier or a substring denotation with nonconstant position values may appear are the right sides of assignments.)

4.3. Character String Assignment

The left and right sides of a character assignment may not share storage. (The assumed implementation of character assignment is to copy characters from the right to the left side.) If the left side is longer than the right, it is padded with blanks. If the left side is shorter than the right, trailing characters are discarded.

4.4. Substrings

It is possible to extract a substring of a character variable or character array element, using the colon notation:

```
a(i, j) (m:n)
```

is the string of $(n-m+1)$ characters beginning at the m^{th} character of the character array element a_{ij}. Results are undefined unless $m \leq n$. Substrings may be used on the left sides of assignments and as procedure actual arguments.

4.5. Exponentiation

It is now permissible to raise real quantities to complex powers, or complex quantities to real or complex powers. (The principal part of the logarithm is used.) Also, multiple exponentiation is now defined:

```
a**b**c 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. (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 B 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 $1 \leq k \leq n$, the return is followed by a branch to the corresponding statement label; otherwise the usual return to the statement following the **call** is executed.

6. Input/Output

6.1. Format Variables

A format may be the value of a character expression (constant or otherwise), or be stored in a character array, as in:

```
write(6, '(i5)') x
```

6.2. END=, ERR=, and IOSTAT= Clauses

A **read** or **write** statement may contain **end=, err=,** and **iostat=** clauses, as in:

```
write(6, 101, err=20, iostat=a(4))
read(5, 101, err=20, end=30, iostat=x)
```

Here 5 and 6 are the *units* on which the I/O is done, 101 is the statement number of the associated format, 20 and 30 are statement numbers, and **a** and **x** are integers. If an error occurs during I/O, control returns to the program at statement 20. If the end of the file is reached, control returns to the program at statement 30. In any case, the variable referred to in the **iostat=** clause is given a value when the I/O statement finishes. (Yes, the value is assigned to the name on the right side of the equal sign.) This value is zero if all went well, negative for end of file, and some positive value for errors.

6.3. Formatted I/O

6.3.1. Character Constants

Character constants in formats are copied literally to the output. It is not allowed to read into character constants or hollerith fields.

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 imbedded 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.

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 Formatting

List-directed I/O is a kind of free form input for sequential I/O. It is invoked by using an asterisk as the format identifier, as in

```
read(6, *) a,b,c
```

On input, values are separated by strings of blanks and possibly a comma. Values, except for character strings, cannot contain blanks. End of record counts as a blank, except in character strings, where it is ignored. Complex constants are given as two real constants separated by a comma and enclosed in parentheses. A null input field, such as between two consecutive commas, means the corresponding variable in the I/O list is not changed. Values may be preceded by repetition counts, as in

```
4*(3.,2.)  2*, 4*'hello'
```

which stands for 4 complex constants, 2 null values, and 4 string constants.

For output, suitable formats are chosen for each item. The values of character strings are printed; they are not enclosed in quotes, so they cannot be read back using list-directed input.

6.6. Direct I/O

A file connected for direct access consists of a set of equal-sized records each of which is uniquely identified by a positive integer. The records may be written or read in any order, using direct access I/O statements.

Direct access **read** and **write** statements have an extra argument, **rec=**, which gives the record number to be read or written.

```
read(2, rec=13, err=20) (a(i), i=1, 203)
```

reads the thirteenth record into the array **a**.

The size of the records must be given by an **open** statement (see below). Direct access files may be connected for either formatted or unformatted I/O.

6.7. Internal Files

Internal files are character string objects, such as variables or substrings, or arrays of type character. In the former cases there is only a single record in the file; in the latter case each array element is a record. The Standard includes only sequential formatted I/O on internal files. (I/O is not a very precise term to use here, but internal files are dealt with using **read** and **write**.) There is no list-directed I/O on internal files. Internal files are used by giving the name of the character object in place of the unit number, as in

```
character*80 x
read(5,'(a)') x
read(x,'(i3,i4)') n1,n2
```

which reads a 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 a compatible extension, direct I/O on internal files. This is like direct I/O on external files, except that the number of records in the file cannot be changed. In this case a record is a single element of an array of character strings.

6.8. OPEN, CLOSE, and INQUIRE Statements

These statements are used to connect and disconnect units and files, and to gather information about units and files.

6.8.1. OPEN

The **open** statement is used to connect a file with a unit, or to alter some properties of the connection. The following is a minimal example.

```
open(1, file='fort.junk')
```

open takes a variety of arguments with meanings described below.

unit= a small non-negative integer which is the unit to which the file is to be connected. We allow, at the time of this writing, 0 through 19. If this parameter is the first one in the **open** statement, the **unit=** can be omitted.

iostat= is the same as in **read** or **write.**

err= is the same as in **read** or **write.**

file= a character expression, which when stripped of trailing blanks, is the name of the file to be connected to the unit. The filename should not be given if the **status='scratch'.**

status= one of **'old', 'new', 'scratch',** or **'unknown'.** If this parameter is not given, **'unknown'** is assumed. 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.

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, namexx)
inquire(file='junk', number=n, exist=l)
```

file= a character variable specifies the file the **inquire** is about. Trailing blanks in the file name are ignored.

unit= an integer variable specifies the unit the **inquire** is about. Exactly one of **file=** or **unit=** must be used.

iostat=, err= are as before.

exist= a logical variable. The logical variable is set to **.true.** if the file or unit exists and is set to **.false.** otherwise.

opened= a logical variable. The logical variable is set to **.true.** if the file is connected to a unit or if the unit is connected to a file, and it is set to **.false.** otherwise.

number= an integer variable to which is assigned the number of the unit connected to the file, if any.

named= a logical variable to which is assigned **.true.** if the file has a name, or **.false.** otherwise.

name= a character variable to which is assigned the name of the file (inquire by file) or the name of the file connected to the unit (inquire by unit). The name will be the full name of the file.

access= a character variable to which will be assigned the value **'sequential'** if the connection is for sequential I/O, **'direct'** if the connection is for direct I/O. The value becomes undefined if there is no connection.

sequential= a character variable to which is assigned the value **'yes'** if the file could be connected for sequential I/O, **'no'** if the file could not be connected for sequential I/O, and **'unknown'** if we can't tell.

direct= a character variable to which is assigned the value **'yes'** if the file could be connected for direct I/O, **'no'** if the file could not be connected for direct I/O, and **'unknown'** if we can't tell.

form= a character variable to which is assigned the value **'unformatted'** if the file is connected for unformatted I/O, **'formatted'** if the file is connected for formatted I/O, or **'print'** for formatted I/O with vertical format control.

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.

The gentle reader will remember that the people who wrote the Standard probably weren't thinking of his needs. Here is an example. The declarations are omitted.

```
open(1, file='/dev/console')
```

On a UNIX system this statement opens the console for formatted sequential I/O. An **inquire** statement for either unit 1 or file "/dev/console" would reveal that the file exists, is connected to unit 1, has a name, namely "/dev/console", is opened for sequential I/O, could be connected for sequential I/O, could not be connected for direct I/O (can't seek), is connected for formatted I/O, could be connected for formatted I/O, could not be connected for

unformatted I/O (can't seek), has neither a record length nor a next record number, and is ignoring blanks in numeric fields.

In the FORTRAN environment, the only way to discover what permissions you have for a file is to open it and try to read and write it. The **err=** parameter will return system error numbers. The **inquire** statement does not give a way of determining permissions.

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.

Introduction to the f77 I/O Library

David L. Wasley

University of California, Berkeley
Berkeley, California 94720

ABSTRACT

The f77 I/O Library implements ANSI 1978 FORTRAN standard input and output with a few minor exceptions. Where the standard is vague, we have tried to provide flexibility within the constraints of the UNIX† operating system. The I/O Library was written originally by Peter J. Weinberger at Bell Labs. A number of logical extensions and enhancements have been provided by this author.

April 1983

† UNIX is a trademark of Bell Laboratories.

Introduction to the f77 I/O Library

David L. Wasley

University of California, Berkeley
Berkeley, California 94720

The f77 I/O library, libI77.a, includes routines to perform all of the standard types of FORTRAN input and output. Several enhancements and extensions to FORTRAN I/O have been added. The f77 library routines use the C stdio library routines to provide efficient buffering for file I/O.

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

There are three forms of I/O: **formatted, unformatted,** and **list-directed.** The last is related to formatted but does not obey all the rules for formatted I/O. There are two modes of access to **external** and **internal** files: **direct** and **sequential.** The definition of a logical record depends upon the combination of I/O form and mode specified by the FORTRAN I/O statement.

1.1.1. Direct access

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

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 I/O

Logical record length for **list-directed** 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.

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** I/O is 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. Both the above will be caught by the compiler. All other flavors of I/O are allowed, although some are not part of the ANSI standard.

Any error detected during I/O processing will cause the program to abort unless alternative action has been provided specifically in the program. Any I/O statement may include an **err=** clause (and **iostat=** clause) to specify an alternative branch to be taken on errors (and return the specific error code). Read statements may include **end=** to branch on end-of-file. 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

The maximum number of logical units that a program may have open at one time is the same as the UNIX system limit, currently 20. Unit numbers must be in the range 0 – 19 because they are used to index an internal control table.

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. No vertical format control is recognized for **direct formatted** output or **list directed** output.

2.4. The open statement

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. Any other **status=** specifier without an associated file name results in opening a file named "fort.N" where N is the specified logical unit number.

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 *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 are parsed at the beginning of each execution of a formatted I/O statement. 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. This occurs in **F** and **G** format conversions. This was not done for **E** and **D** since the embedded blanks in the external datum causes problems for other input systems.

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 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 **Ew.dEe** format specifier has been extended to **D** also. The form Ew.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 $2 \leq n \leq 36$. If *n* is omitted, the default decimal radix is restored.

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 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, 16R, 8I10.8 )
```

Note: Unsigned integer values greater than (2**30 - 1), i.e. any signed negative value, can not be read by FORTRAN input routines. All internal values will be output correctly.

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. 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 input of a string not enclosed in quotes. The string must not start with a digit, and can not contain a separator (, or /) or blank (space 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.

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. Preattachment of logical units

The *ioinit* routine also can be used to attach logical units to specific files at run time. 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/libI77.a ioinit.f".

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 *tapeio*(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 if the **err=** return is taken. 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 (), or an extremely long format statement.

/* 101 */ "illegal unit number"
It is illegal to close logical unit 0. Negative unit numbers are not allowed. The upper limit is system dependent.

/* 102 */ "formatted io not allowed"
The logical unit was opened for unformatted I/O.

/* 103 */ "unformatted io not allowed"
The logical unit was opened for formatted I/O.

/* 104 */ "direct io not allowed"
The logical unit was opened for sequential access, or the logical record length was specified as 0.

/* 105 */ "sequential io 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 */ "read unexpected character"
Certain format conversions can't tolerate non-numeric data. Logical data must be T or F.

/* 116 */ "blank logical input field"

/* 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 */ "unknown system error"
Shouldn't happen, but .....

/* 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.

Appendix B

Exceptions to the ANSI Standard

A few exceptions to the ANSI standard remain.

1) Vertical format control

The "+" carriage control specifier is not implemented. 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.

2) Default files

Files created by default use of **rewind** or **endfile** statements are opened for **sequential formatted** access. There is no way to redefine such a file to allow **direct** or **unformatted** access.

3) 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.

4) 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.

Berkeley Pascal User's Manual
Version 3.1 – April 1986

William N. Joy‡, Susan L. Graham, Charles B. Haley‡,
Marshall Kirk McKusick, and Peter B. Kessler‡

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

ABSTRACT

Berkeley Pascal is designed for interactive instructional use and runs on the PDP/11 and VAX/11 computers. Interpretive code is produced, providing fast translation at the expense of slower execution speed. There is also a fully compatible compiler for the VAX/11. An execution profiler and Wirth's cross reference program are also available with the system.

The system supports full Pascal. The language accepted is 'standard' Pascal, and a small number of extensions. There is an option to suppress the extensions. The extensions include a separate compilation facility and the ability to link to object modules produced from other source languages.

The *User's Manual* gives a list of sources relating to the UNIX† system, the Pascal language, and the Berkeley Pascal system. Basic usage examples are provided for the Pascal components *pi*, *px*, *pix*, *pc*, and *pxp*. Errors commonly encountered in these programs are discussed. Details are given of special considerations due to the interactive implementation. A number of examples are provided including many dealing with input/output. An appendix supplements Wirth's *Pascal Report* to form the full definition of the Berkeley implementation of the language.

‡Author's current addresses: William Joy: Sun Microsystems, 2550 Garcia Ave., Mountain View, CA 94043; Charles Haley: S & B Associates, 1110 Centennial Ave., Piscataway, NJ 08854; Peter Kessler: Xerox Research Park, Palo Alto, CA
† UNIX is a trademark of Bell Laboratories.

Introduction

The Berkeley Pascal *User's Manual* consists of five major sections and an appendix. In section 1 we give sources of information about UNIX, about the programming language Pascal, and about the Berkeley implementation of the language. Section 2 introduces the Berkeley implementation and provides a number of tutorial examples. Section 3 discusses the error diagnostics produced by the translators *pc* and *pi*, and the runtime interpreter *px*. Section 4 describes input/output with special attention given to features of the interactive implementation and to features unique to UNIX. Section 5 gives details on the components of the system and explanation of all relevant options. The *User's Manual* concludes with an appendix to Wirth's *Pascal Report* with which it forms a precise definition of the implementation.

History of the implementation

The first Berkeley system was written by Ken Thompson in early 1976. The main features of the present system were implemented by Charles Haley and William Joy during the latter half of 1976. Earlier versions of this system have been in use since January, 1977.

The system was moved to the VAX-11 by Peter Kessler and Kirk McKusick with the porting of the interpreter in the spring of 1979, and the implementation of the compiler in the summer of 1980.

1. Sources of information

This section lists the resources available for information about general features of UNIX, text editing, the Pascal language, and the Berkeley Pascal implementation, concluding with a list of references. The available documents include both so-called standard documents – those distributed with all UNIX system – and documents (such as this one) written at Berkeley.

1.1. Where to get documentation

Current documentation for most of the UNIX system is available "on line" at your terminal. Details on getting such documentation interactively are given in section 1.3.

1.2. Documentation describing UNIX

The following documents are those recommended as tutorial and reference material about the UNIX system. We give the documents with the introductory and tutorial materials first, the reference materials last.

UNIX For Beginners – Second Edition

This document is the basic tutorial for UNIX available with the standard system.

Communicating with UNIX

This is also a basic tutorial on the system and assumes no previous familiarity with computers; it was written at Berkeley.

An introduction to the C shell

This document introduces *csh*, the shell in common use at Berkeley, and provides a good deal of general description about the way in which the system functions. It provides a useful glossary of terms used in discussing the system.

UNIX Programmer's Manual

This manual is the major source of details on the components of the UNIX system. It consists of an Introduction, a permuted index, and eight command sections. Section 1 consists of descriptions of most of the "commands" of UNIX. Most of the other sections have limited relevance to the user of Berkeley Pascal, being of interest mainly to system programmers.

UNIX documentation often refers the reader to sections of the manual. Such a reference consists of a command name and a section number or name. An example of such a reference would be: *ed* (1). Here *ed* is a command name – the standard UNIX text editor, and '(1)' indicates that its documentation is in section 1 of the manual.

The pieces of the Berkeley Pascal system are *pi* (1), *px* (1), the combined Pascal translator and interpretive executor *pix* (1), the Pascal compiler *pc* (1), the Pascal execution profiler *pxp* (1), and the Pascal cross-reference generator *pxref* (1).

It is possible to obtain a copy of a manual section by using the *man* (1) command. To get the Pascal documentation just described one could issue the command:

% man pi

to the shell. The user input here is shown in **bold face**; the '% ', which was printed by the shell as a prompt, is not. Similarly the command:

% man man

asks the *man* command to describe itself.

1.3. Text editing documents

The following documents introduce the various UNIX text editors. Most Berkeley users use a version of the text editor *ex;* either *edit,* which is a version of *ex* for new and casual users, *ex* itself, or *vi* (visual) which focuses on the display editing portion of *ex.*

A Tutorial Introduction to the UNIX Text Editor

This document, written by Brian Kernighan of Bell Laboratories, is a tutorial for the standard UNIX text editor *ed.* It introduces you to the basics of text editing, and provides enough information to meet day-to-day editing needs, for *ed* users.

Edit: A tutorial

This introduces the use of *edit,* an editor similar to *ed* which provides a more hospitable environment for beginning users.

Ex/edit Command Summary

This summarizes the features of the editors *ex* and *edit* in a concise form. If you have used a line oriented editor before this summary alone may be enough to get you started.

Ex Reference Manual – Version 3.7

A complete reference on the features of *ex* and *edit.*

An Introduction to Display Editing with Vi

Vi is a display oriented text editor. It can be used on most any CRT terminal, and uses the screen as a window into the file you are editing. Changes you make to the file are reflected in what you see. This manual serves both as an introduction to editing with *vi* and a reference manual.

Vi Quick Reference

This reference card is a handy quick guide to *vi;* you should get one when you get the introduction to *vi.*

1.4. Pascal documents – The language

This section describes the documents on the Pascal language which are likely to be most useful to the Berkeley Pascal user. Complete references for these documents are given in section 1.7.

Pascal User Manual

By Kathleen Jensen and Niklaus Wirth, the *User Manual* provides a tutorial introduction to the features of the language Pascal, and serves as an excellent quick-reference to the language. The reader with no familiarity with Algol-like languages may prefer one of the Pascal text books listed below, as they provide more examples and explanation. Particularly important here are pages 116-118 which define the syntax of the language. Sections 13 and 14 and Appendix F pertain only to the 6000-3.4 implementation of Pascal.

Pascal Report

By Niklaus Wirth, this document is bound with the *User Manual.* It is the guiding reference for implementors and the fundamental definition of the language. Some programmers find this report too concise to be of practical use, preferring the *User Manual* as a reference.

Books on Pascal

Several good books which teach Pascal or use it as a medium are available. The books by Wirth *Systematic Programming* and *Algorithms + Data Structures = Programs* use Pascal as a vehicle for teaching programming and data structure concepts respectively. They are both recommended. Other books on Pascal are listed in the references below.

1.5. Pascal documents – The Berkeley Implementation

This section describes the documentation which is available describing the Berkeley implementation of Pascal.

User's Manual

The document you are reading is the *User's Manual* for Berkeley Pascal. We often refer the reader to the Jensen-Wirth *User Manual* mentioned above, a different document with a similar name.

Manual sections

The sections relating to Pascal in the *UNIX Programmer's Manual* are *pix* (1), *pi* (1), *pc* (1), *px* (1), *pxp* (1), and *pxref* (1). These sections give a description of each program, summarize the available options, indicate files used by the program, give basic information on the diagnostics produced and include a list of known bugs.

Implementation notes

For those interested in the internal organization of the Berkeley Pascal system there are a series of *Implementation Notes* describing these details. The *Berkeley Pascal PXP Implementation Notes* describe the Pascal interpreter *px* ; and the *Berkeley Pascal PX Implementation Notes* describe the structure of the execution profiler *pxp* .

1.6. References

UNIX Documents

Communicating With UNIX
Computer Center
University of California, Berkeley
January, 1978.

Ricki Blau and James Joyce
Edit: a tutorial
UNIX User's Supplementary Documents (USD), 14
University of California, Berkeley, CA. 94720
April, 1986.

Ex/edit Command Summary
Computer Center
University of California, Berkeley
August, 1978.

William Joy
Ex Reference Manual - Version 3.7
UNIX User's Supplementary Documents (USD), 16
University of California, Berkeley, CA. 94720
April, 1986.

William Joy
An Introduction to Display Editing with Vi
UNIX User's Supplementary Documents (USD), 15
University of California, Berkeley, CA. 94720
April, 1986.

William Joy
An Introduction to the C shell (Revised)
UNIX User's Supplementary Documents (USD), 4
University of California, Berkeley, CA. 94720
April, 1986.

Brian W. Kernighan
UNIX for Beginners - Second Edition

UNIX User's Supplementary Documents (USD), 1
University of California, Berkeley, CA. 94720
April, 1986.

Brian W. Kernighan
A Tutorial Introduction to the UNIX Text Editor
UNIX User's Supplementary Documents (USD), 12
University of California, Berkeley, CA. 94720
April, 1986.

Dennis M. Ritchie and Ken Thompson
The UNIX Time Sharing System
Reprinted from Communications of the ACM July 1974 in
UNIX Programmer's Supplementary Documents, Volume 2 (PS2), 1
University of California, Berkeley, CA. 94720
April, 1986.

Pascal Language Documents

Cooper and Clancy
Oh! Pascal!, 2nd Edition
W. W. Norton & Company, Inc.
500 Fifth Ave., NY, NY. 10110
1985, 475 pp.

Cooper
Standard Pascal User Reference Manual
W. W. Norton & Company, Inc.
500 Fifth Ave., NY, NY. 10110
1983, 176 pp.

Kathleen Jensen and Niklaus Wirth
Pascal - User Manual and Report
Springer-Verlag, New York.
1975, 167 pp.

Niklaus Wirth
Algorithms + Data structures = Programs
Prentice-Hall, New York.
1976, 366 pp.

Berkeley Pascal documents

The following documents are available from the Computer Center Library at the University of California, Berkeley.

William N. Joy
Berkeley Pascal PX Implementation Notes
Version 1.1, April 1979.
(Vax-11 Version 2.0 By Kirk McKusick, December, 1979)

William N. Joy
Berkeley Pascal PXP Implementation Notes
Version 1.1, April 1979.

2. Basic UNIX Pascal

The following sections explain the basics of using Berkeley Pascal. In examples here we use the text editor *ex* (1). Users of the text editor *ed* should have little trouble following these examples, as *ex* is similar to *ed*. We use *ex* because it allows us to make clearer examples.† The new UNIX user will find it helpful to read one of the text editor documents described in section 1.4 before continuing with this section.

2.1. A first program

To prepare a program for Berkeley Pascal we first need to have an account on UNIX and to 'login' to the system on this account. These procedures are described in the documents *Communicating with UNIX* and *UNIX for Beginners*.

Once we are logged in we need to choose a name for our program; let us call it 'first' as this is the first example. We must also choose a name for the file in which the program will be stored. The Berkeley Pascal system requires that programs reside in files which have names ending with the sequence '.p' so we will call our file 'first.p'.

A sample editing session to create this file would begin:

```
% ex first.p
"first.p" [New file]
:
```

We didn't expect the file to exist, so the error diagnostic doesn't bother us. The editor now knows the name of the file we are creating. The ':' prompt indicates that it is ready for command input. We can add the text for our program using the 'append' command as follows.

```
:append
program first(output)
begin
    writeln('Hello, world!')
end.
.
:
```

The line containing the single '.' character here indicated the end of the appended text. The ':' prompt indicates that *ex* is ready for another command. As the editor operates in a temporary work space we must now store the contents of this work space in the file 'first.p' so we can use the Pascal translator and executor *pix* on it.

```
:write
"first.p" [New file] 4 lines, 59 characters
:quit
%
```

We wrote out the file from the edit buffer here with the 'write' command, and *ex* indicated the number of lines and characters written. We then quit the editor, and now have a prompt from the shell.‡

---

† Users with CRT terminals should find the editor *vi* more pleasant to use; we do not show its use here because its display oriented nature makes it difficult to illustrate.

‡ Our examples here assume you are using *csh*.

We are ready to try to translate and execute our program.

```
% pix first.p
Tue Oct 14 21:37 1980  first.p:
    2  begin
e -----↑--- Inserted ';'
Execution begins...
Hello, world!
Execution terminated.

1 statements executed in 0.02 seconds cpu time.
%
```

The translator first printed a syntax error diagnostic. The number 2 here indicates that the rest of the line is an image of the second line of our program. The translator is saying that it expected to find a ';' before the keyword **begin** on this line. If we look at the Pascal syntax charts in the Jensen-Wirth *User Manual*, or at some of the sample programs therein, we will see that we have omitted the terminating ';' of the **program** statement on the first line of our program.

One other thing to notice about the error diagnostic is the letter 'e' at the beginning. It stands for 'error', indicating that our input was not legal Pascal. The fact that it is an 'e' rather than an 'E' indicates that the translator managed to recover from this error well enough that generation of code and execution could take place. Execution is possible whenever no fatal 'E' errors occur during translation. The other classes of diagnostics are 'w' warnings, which do not necessarily indicate errors in the program, but point out inconsistencies which are likely to be due to program bugs, and 's' standard-Pascal violations.†

After completing the translation of the program to interpretive code, the Pascal system indicates that execution of the translated program began. The output from the execution of the program then appeared. At program termination, the Pascal runtime system indicated the number of statements executed, and the amount of cpu time used, with the resolution of the latter being 1/60'th of a second.

Let us now fix the error in the program and translate it to a permanent object code file *obj* using *pi*. The program *pi* translates Pascal programs but stores the object code instead of executing it‡.

```
% ex first.p
"first.p" 4 lines, 59 characters
:1 print
program first(output)
:s/$/;
program first(output);
:write
"first.p" 4 lines, 60 characters
:quit
% pi first.p
%
```

If we now use the UNIX *ls* list files command we can see what files we have:

---

†The standard Pascal warnings occur only when the associated s translator option is enabled. The s option is discussed in sections 5.1 and A.6 below. Warning diagnostics are discussed at the end of section 3.2, the associated w option is described in section 5.2.

‡This script indicates some other useful approaches to debugging Pascal programs. As in *ed* we can shorten commands in *ex* to an initial prefix of the command name as we did with the *substitute* command here. We have also used the '!' shell escape command here to execute other commands with a shell without leaving the editor.

```
% ls
first.p
obj
%
```

The file 'obj' here contains the Pascal interpreter code. We can execute this by typing:

```
% px obj
Hello, world!

1 statements executed in 0.02 seconds cpu time.
%
```

Alternatively, the command:

```
% obj
```

will have the same effect. Some examples of different ways to execute the program follow.

```
% px
Hello, world!

1 statements executed in 0.02 seconds cpu time.
% pi -p first.p
% px obj
Hello, world!
% pix -p first.p
Hello, world!
%
```

Note that *px* will assume that 'obj' is the file we wish to execute if we don't tell it otherwise. The last two translations use the **-p** no-post-mortem option to eliminate execution statistics and 'Execution begins' and 'Execution terminated' messages. See section 5.2 for more details. If we now look at the files in our directory we will see:

```
% ls
first.p
obj
%
```

We can give our object program a name other than 'obj' by using the move command *mv* (1). Thus to name our program 'hello':

```
% mv obj hello
% hello
Hello, world!
% ls
first.p
hello
%
```

Finally we can get rid of the Pascal object code by using the *rm* (1) remove file command, e.g.:

```
% rm hello
% ls
first.p
%
```

For small programs which are being developed *pix* tends to be more convenient to use than *pi* and *px*. Except for absence of the *obj* file after a *pix* run, a *pix* command is equivalent to a *pi* command followed by a *px* command. For larger programs, where a number of runs testing different parts of the program are to be made, *pi* is useful as this *obj* file can be executed any desired number of times.

2.2. A larger program

Suppose that we have used the editor to put a larger program in the file 'bigger.p'. We can list this program with line numbers by using the program *cat*-n i.e.:

```
% cat -n bigger.p
%
```

This program is similar to program 4.9 on page 30 of the Jensen-Wirth *User Manual*. A number of problems have been introduced into this example for pedagogical reasons.

If we attempt to translate and execute the program using *pix* we get the following response:

```
% pix bigger.p
Tue Oct 14 21:37 1980  bigger.p:
    9        h = 34;       (* Character position of x-axis *)
w ----------------------------↑ ------ (* in a (* ... *) comment
   16        for i := 0 to lim begin
e ---------------------------------↑ ------ Inserted keyword do
   18              y := exp(-x9 * sin(i * x);
E ---------------------------------↑ ------ Undefined variable
e ------------------------------------------------↑ ------ Inserted ')'
   19              n := Round(s * y) + h;
E ------------------------↑ ------ Undefined function
E ------------------------------------------↑ ------ Undefined variable
   23          .   writeln('*')
e -------------------↑ ------ Inserted ';'
   24  end.
E -----↑ ------ Expected keyword until
E ----------↑ ------ Unexpected end-of-file – QUIT
Execution suppressed due to compilation errors
%
```

Since there were fatal 'E' errors in our program, no code was generated and execution was necessarily suppressed. One thing which would be useful at this point is a listing of the program with the error messages. We can get this by using the command:

```
% pi -l bigger.p
```

There is no point in using *pix* here, since we know there are fatal errors in the program. This command will produce the output at our terminal. If we are at a terminal which does not produce a hard copy we may wish to print this listing off-line on a line printer. We can do this with the command:

```
% pi -l bigger.p | lpr
```

In the next few sections we will illustrate various aspects of the Berkeley Pascal system by correcting this program.

2.3. Correcting the first errors

Most of the errors which occurred in this program were *syntactic* errors, those in the format and structure of the program rather than its content. Syntax errors are flagged by printing the offending line, and then a line which flags the location at which an error was detected. The flag line also gives an explanation stating either a possible cause of the error, a simple action which can be taken to recover from the error so as to be able to continue the analysis, a symbol which was expected at the point of error, or an indication that the input was 'malformed'. In the last case, the recovery may skip ahead in the input to a point where analysis of the program can continue.

In this example, the first error diagnostic indicates that the translator detected a comment within a comment. While this is not considered an error in 'standard' Pascal, it usually corresponds to an error in the program which is being translated. In this case, we have accidentally omitted the trailing '*)' of the comment on line 8. We can begin an editor session to correct this problem by doing:

```
% ex bigger.p
"bigger.p" 24 lines, 512 characters
:8s/$/ *)
        s = 32;         (* 32 character width for interval [x, x+1] *)
:
```

The second diagnostic, given after line 16, indicates that the keyword **do** was expected before the keyword **begin** in the **for** statement. If we examine the *statement* syntax chart on page 118 of the Jensen-Wirth *User Manual* we will discover that **do** is a necessary part of the **for** statement. Similarly, we could have referred to section C.3 of the Jensen-Wirth *User Manual* to learn about the **for** statement and gotten the same information there. It is often useful to refer to these syntax charts and to the relevant sections of this book.

We can correct this problem by first scanning for the keyword **for** in the file and then substituting the keyword **do** to appear in front of the keyword **begin** there. Thus:

```
:/for
        for i := 0 to lim begin
:s/begin/do &
        for i := 0 to lim do begin
:
```

The next error in the program is easy to pinpoint. On line 18, we didn't hit the shift key and got a '9' instead of a ')'. The translator diagnosed that 'x9' was an undefined variable and, later, that a ')' was missing in the statement. It should be stressed that *pi* is not suggesting that you should insert a ')' before the ';'. It is only indicating that making this change will help it to be able to

continue analyzing the program so as to be able to diagnose further errors. You must then determine the true cause of the error and make the appropriate correction to the source text.

This error also illustrates the fact that one error in the input may lead to multiple error diagnostics. *Pi* attempts to give only one diagnostic for each error, but single errors in the input sometimes appear to be more than one error. It is also the case that *pi* may not detect an error when it occurs, but may detect it later in the input. This would have happened in this example if we had typed 'x' instead of 'x9'.

The translator next detected, on line 19, that the function *Round* and the variable *h* were undefined. It does not know about *Round* because Berkeley Pascal normally distinguishes between upper and lower case.† On UNIX lower-case is preferred‡, and all keywords and built-in **procedure** and **function** names are composed of lower-case letters, just as they are in the Jensen-Wirth *Pascal Report*. Thus we need to use the function *round* here. As far as *h* is concerned, we can see why it is undefined if we look back to line 9 and note that its definition was lost in the non-terminated comment. This diagnostic need not, therefore, concern us.

The next error which occurred in the program caused the translator to insert a ';' before the statement calling *writeln* on line 23. If we examine the program around the point of error we will see that the actual error is that the keyword **until** and an associated expression have been omitted here. Note that the diagnostic from the translator does not indicate the actual error, and is somewhat misleading. The translator made the correction which seemed to be most plausible. As the omission of a ';' character is a common mistake, the translator chose to indicate this as a possible fix here. It later detected that the keyword **until** was missing, but not until it saw the keyword **end** on line 24. The combination of these diagnostics indicate to us the true problem.

The final syntactic error message indicates that the translator needed an **end** keyword to match the **begin** at line 15. Since the **end** at line 24 is supposed to match this **begin**, we can infer that another **begin** must have been mismatched, and have matched this **end**. Thus we see that we need an **end** to match the **begin** at line 16, and to appear before the final **end**. We can make these corrections:

```
:/x9/s//x)
            y := exp(-x) * sin(i * x);
:+s/Round/round
            n := round(s * y) + h;
:/write
                  write(' ');
:/
            writeln('*')
:insert
            until n = 0;
.
:$
end.
:insert
      end
.
:
```

†In "standard" Pascal no distinction is made based on case.
‡One good reason for using lower-case is that it is easier to type.

At the end of each **procedure** or **function** and the end of the **program** the translator summarizes references to undefined variables and improper usages of variables. It also gives warnings about potential errors. In our program, the summary errors do not indicate any further problems but the warning that *c* is unused is somewhat suspicious. Examining the program we see that the constant was intended to be used in the expression which is an argument to ***sin***, so we can correct this expression, and translate the program. We have now made a correction for each diagnosed error in our program.

```
:?i ?s//c /
            y := exp(−x) * sin(c * x);
:write
"bigger.p" 26 lines, 538 characters
:quit
% pi bigger.p
%
```

It should be noted that the translator suppresses warning diagnostics for a particular **procedure**, **function** or the main **program** when it finds severe syntax errors in that part of the source text. This is to prevent possibly confusing and incorrect warning diagnostics from being produced. Thus these warning diagnostics may not appear in a program with bad syntax errors until these errors are corrected.

We are now ready to execute our program for the first time. We will do so in the next section after giving a listing of the corrected program for reference purposes.

```
% cat –n bigger.p
     1  (*
     2   * Graphic representation of a function
     3   *    f(x) = exp(–x) * sin(2 * pi * x)
     4   *)
     5  program graph1(output);
     6  const
     7            d = 0.0625;   (* 1/16, 16 lines for interval [x, x+1] *)
     8            s = 32;       (* 32 character width for interval [x, x+1] *)
     9            h = 34;       (* Character position of x–axis *)
    10            c = 6.28138;  (* 2 * pi *)
    11            lim = 32;
    12  var
    13            x, y: real;
    14            i, n: integer;
    15  begin
    16            for i := 0 to lim do begin
    17                      x := d / i;
    18                      y := exp(–x) * sin(c * x);
    19                      n := round(s * y) + h;
    20                      repeat
    21                                write(' ');
    22                                n := n – 1
    23                      until n = 0;
    24                      writeln('*')
    25            end
    26  end.
%
```

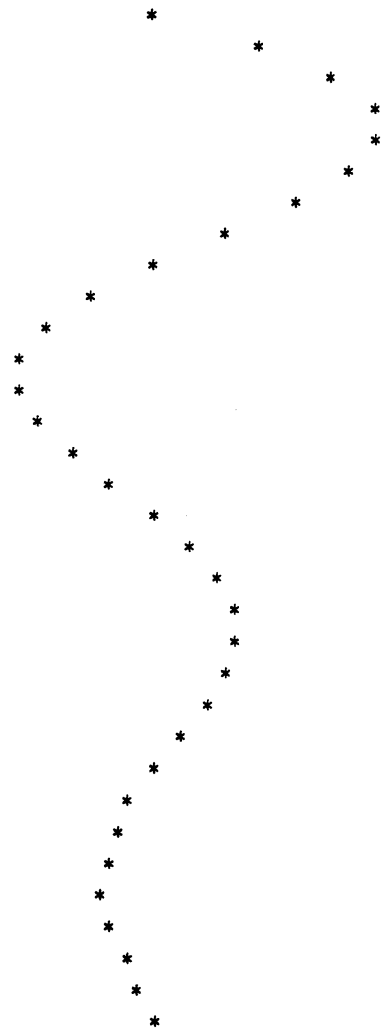
2.4. Executing the second example

We are now ready to execute the second example. The following output was produced by our first run.

```
% px
Execution begins...


Floating point division error

      Error in "graph1"+2 near line 17.
Execution terminated abnormally.

2 statements executed in 0.05 seconds cpu time.
%
```

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:

```
% ex bigger.p
"bigger.p" 26 lines, 538 characters
:17
      x := d / i
:s'/'*
      x := d * i
:write
"bigger.p" 26 lines, 538 characters
:q
% pix bigger.p
Execution begins...
```



```
Execution terminated.

2550 statements executed in 0.30 seconds cpu time.
%
```

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.

```
% px | lpr
Execution begins...
Execution terminated.

2550 statements executed in 0.37 seconds cpu time.
%
```

Note here 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 rather than '|', i.e.:

```
% 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:

```
% pi -p bigger.p
% px | lpr
%
```

This option also disables the statement limit which normally guards against infinite looping. You should not use it until your program is debugged. Also if **p** is specified and an error occurs, you will not get run time diagnostic information to help you determine what the problem is.

2.5. Formatting the program listing

It is possible to use special lines within the source text of a program to format the program listing. An empty line (one with no characters on it) corresponds to a 'space' macro in an assembler, leaving a completely blank line without a line number. A line containing only a control-l (form-feed) character will cause a page eject in the listing with the corresponding line number suppressed. This corresponds to an 'eject' pseudo-instruction. See also section 5.2 for details on the **n** and **i** options of *pi*.

2.6. Execution profiling

An execution profile consists of a structured listing of (all or part of) a program with information about the number of times each statement in the program was executed for a particular run of the program. These profiles can be used for several purposes. In a program which was

abnormally terminated due to excessive looping or recursion or by a program fault, the counts can facilitate location of the error. Zero counts mark portions of the program which were not executed; during the early debugging stages they should prompt new test data or a re-examination of the program logic. The profile is perhaps most valuable, however, in drawing attention to the (typically small) portions of the program that dominate execution time. This information can be used for source level optimization.

An example

A prime number is a number which is divisible only by itself and the number one. The program *primes*, written by Niklaus Wirth, determines the first few prime numbers. In translating the program we have specified the **z** option to *pix*. This option causes the translator to generate counters and count instructions sufficient in number to determine the number of times each statement in the program was executed.† When execution of the program completes, either normally or abnormally, this count data is written to the file *pmon.out* in the current directory.‡ It is then possible to prepare an execution profile by giving *pxp* the name of the file associated with this data, as was done in the following example.

```
% pix -l -z primes.p
Berkeley Pascal PI -- Version 2.0 (Sat Oct 18 21:01:54 1980)

Tue Oct 14 21:38 1980  primes.p

 1  program primes(output);
 2  const n = 50; n1 = 7; (*n1 = sqrt(n)*)
 3  var i,k,x,inc,lim,square,l: integer;
 4      prim: boolean;
 5      p,v: array[1..n1] of integer;
 6  begin
 7      write(2:6, 3:6); l := 2;
 8      x := 1; inc := 4; lim := 1; square := 9;
 9      for i := 3 to n do
10      begin (*find next prime*)
11          repeat x := x + inc; inc := 6-inc;
12              if square <= x then
13                  begin lim := lim+1;
14                      v[lim] := square; square := sqr(p[lim+1])
15                  end ;
16              k := 2; prim := true;
17              while prim and (k<lim) do
18              begin k := k+1;
19                  if v[k] < x then v[k] := v[k] + 2*p[k];
20                  prim := x <> v[k]
21              end
22          until prim;
23          if i <= n1 then p[i] := x;
24          write(x:6); l := l+1;
25          if l = 10 then
```

†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.

```
    26              begin writeln; l := 0
    27              end
    28      end ;
    29      writeln;
    30  end .
Execution begins...
     2     3     5     7    11    13    17    19    23    29
    31    37    41    43    47    53    59    61    67    71
    73    79    83    89    97   101   103   107   109   113
   127   131   137   139   149   151   157   163   167   173
   179   181   191   193   197   199   211   223   227   229

Execution terminated.

1404 statements executed in 0.17 seconds cpu time.
%
```

Discussion

The header lines of the outputs of *pix* and *pxp* in this example indicate the version of the translator and execution profiler in use at the time this example was prepared. The time given with the file name (also on the header line) indicates the time of last modification of the program source file. This time serves to *version stamp* the input program. *Pxp* also indicates the time at which the profile data was gathered.

```
% pxp -z primes.p
Berkeley Pascal PXP -- Version 1.1 (May 7, 1979)

Tue Oct 14 21:38 1980  primes.p

Profiled Tue Oct 21 18:48 1980

     1          1.-----program primes(output);
     2                 const
     2                 |   n = 50;
     2                 |   n1 = 7; (*n1 = sqrt(n)*)
     3                 var
     3                 |   i, k, x, inc, lim, square, l: integer;
     4                 |   prim: boolean;
     5                 |   p, v: array [1..n1] of integer;
     6                 begin
     7                 |   write(2: 6, 3: 6);
     7                 |   l := 2;
     8                 |   x := 1;
     8                 |   inc := 4;
     8                 |   lim := 1;
     8                 |   square := 9;
     9                 |   for i := 3 to n do begin (*find next prime*)
     9                48.-----|   repeat
```

```
11              76.-----|   x := x + inc;
11                      |   inc := 6 - inc;
12                      |   if square <= x then begin
13                   5.-----|   lim := lim + 1;
14                           |   v[lim] := square;
14                           |   square := sqr(p[lim + 1])
14                      |   end;
16                      |   k := 2;
16                      |   prim := true;
17                      |   while prim and (k < lim) do begin
18                 157.-----|   k := k + 1;
19                           |   if v[k] < x then
19                        42.-----|   v[k] := v[k] + 2 * p[k];
20                          |   prim := x <> v[k]
20                      |   end
20                      until prim;
23                |   if i <= n1 then
23                5.-----|   p[i] := x;
24                |   write(x: 6);
24                |   l := l + 1;
25                |   if l = 10 then begin
26                5.-----|   writeln;
26                       |   l := 0
26                |   end
26              |   end;
29              |   writeln
29              end.
%
```

To determine the number of times a statement was executed, one looks to the left of the statement and finds the corresponding vertical bar '|'. If this vertical bar is labelled with a count then that count gives the number of times the statement was executed. If the bar is not labelled, we look up in the listing to find the first '|' which directly above the original one which has a count and that is the answer. Thus, in our example, *k* was incremented 157 times on line 18, while the *write* procedure call on line 24 was executed 48 times as given by the count on the **repeat**.

More information on *pxp* can be found in its manual section *pxp* (1) and in sections 5.4, 5.5 and 5.10.

3. Error diagnostics

This section of the *User's Manual* discusses the error 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 more details. All the diagnostics given by *pi* will also be given by *pc*.

3.1. Translator syntax errors

A few comments on the general nature of the syntax errors usually made by Pascal programmers and the recovery mechanisms of the current translator may help in using the system.

Illegal characters

Characters such as '$', '!', and '@' are not part of the language Pascal. If they are found in the source program, and are not part of a constant string, a constant character, or a comment, they are considered to be 'illegal characters'. This can happen if you leave off an opening string quote '. Note that the character ", although used in English to quote strings, is not used to quote strings in Pascal. Most non-printing characters in your input are also illegal except in character constants and character strings. Except for the tab and form feed characters, which are used to ease formatting of the program, non-printing characters in the input file print as the character '?' so that they will show in your listing.

String errors

There is no character string of length 0 in Pascal. Consequently the input '' is not acceptable. Similarly, encountering an end-of-line after an opening string quote ' without encountering the matching closing quote yields the diagnostic "Unmatched ' for string". It is permissible to use the character '#' instead of ' to delimit character and constant strings for portability reasons. For this reason, a spuriously placed '#' sometimes causes the diagnostic about unbalanced quotes. Similarly, a '#' in column one is used when preparing programs which are to be kept in multiple files. See section 5.11 for details.

Comments in a comment, non-terminated comments

As we saw above, these errors are usually caused by leaving off a comment delimiter. You can convert parts of your program to comments without generating this diagnostic since there are two different kinds of comments – those delimited by '{' and '}', and those delimited by '(*' and '*)'. Thus consider:

```
{ This is a comment enclosing a piece of program
a := functioncall;      (* comment within comment *)
procedurecall;
lhs := rhs;         (* another comment *)
}
```

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 placed within comments.

If a comment does not terminate before the end of the input file, the translator will point to the beginning of the comment, indicating that the comment is not terminated. In this case processing will terminate immediately. See the discussion of "QUIT" below.

---

†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.

Digits in numbers

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:

```
Tue Oct 14 21:37 1980  digits.p:
   4  r := 0.;
e  ------------↑ ----- Digits required after decimal point
   5  r := .0;
e  ----------↑ ----- Digits required before decimal point
   6  r := 1.e10;
e  ------------↑ ----- Digits required after decimal point
   7  r := .05e-10;
e  ----------↑ ----- Digits required before decimal point
```

These same constructs are also illegal as input to the Pascal interpreter *px*.

Replacements, insertions, and deletions

When a syntax error is encountered in the input text, the parser invokes an error recovery procedure. This procedure examines the input text immediately after the point of error and considers a set of simple corrections to see whether they will allow the analysis to continue. These corrections involve replacing an input token with a different token, inserting a token, or replacing an input token with a different token. Most of these changes will not cause fatal syntax errors. The exception is the insertion of or replacement with a symbol such as an identifier or a number; in this case the recovery makes no attempt to determine *which* identifier or *what* number should be inserted, hence these are considered fatal syntax errors.

Consider the following example.

```
% pix -l synerr.p
Berkeley Pascal PI -- Version 2.0 (Sat Oct 18 21:01:54 1980)

Tue Oct 21 23:51 1980  synerr.p

   1  program syn(output);
   2  var i, j are integer;
e  --------------↑--- Replaced identifier with a ':'
   3  begin
   4          for j :* 1 to 20 begin
e  --------------------↑--- Replaced '*' with a '='
e  ------------------------------↑--- Inserted keyword do
   5                  write(j);
   6                  i = 2 ** j;
e  ----------------------↑--- Inserted ':'
E  --------------------------↑--- Inserted identifier
   7                  writeln(i))
E  ------------------------------↑--- Deleted ')'
   8          end
   9  end.
```

```
%
```

The only surprise here may be that Pascal does not have an exponentiation operator, hence the complaint about '**'. This error illustrates that, if you assume that the language has a feature which it does not, the translator diagnostic may not indicate this, as the translator is unlikely to recognize the construct you supply.

Undefined or improper identifiers

If an identifier is encountered in the input but is undefined, the error recovery will replace it with an identifier of the appropriate class. Further references to this identifier will be summarized at the end of the containing **procedure** or **function** or at the end of the **program** if the reference occurred in the main program. Similarly, if an identifier is used in an inappropriate way, e.g. if a **type** identifier is used in an assignment statement, or if a simple variable is used where a **record** variable is required, a diagnostic will be produced and an identifier of the appropriate type inserted. Further incorrect references to this identifier will be flagged only if they involve incorrect use in a different way, with all incorrect uses being summarized in the same way as undefined variable uses are.

Expected symbols, malformed constructs

If none of the above mentioned corrections appear reasonable, the error 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 diagnostic which indicates that the given symbol was 'Expected'.

In cases where none of these corrections resolve the problems in the input, the recovery may issue a diagnostic that indicates that the input is "malformed". If necessary, the translator may then skip forward in the input to a place where analysis can continue. This process may cause some errors in the text to be missed.

Consider the following example:

```
% pix -l synerr2.p
Berkeley Pascal PI -- Version 2.0 (Sat Oct 18 21:01:54 1980)

Tue Oct 14 21:38 1980  synerr2.p

    1  program synerr2(input,outpu);
    2  integer a(10)
E -----↑------ Malformed declaration
    3  begin
    4          read(b);
E ------------------↑------ Undefined variable
    5          for c := 1 to 10 do
E ----------------↑------ Undefined variable
    6                  a(c) := b * c;
E --------------------↑------ Undefined procedure
E ------------------------↑------ Malformed statement
    7  end.
E 1 - File outpu listed in program statement but not declared
e 1 - The file output must appear in the program statement file list
```

```
In program synerr2:
  E – a undefined on line 6
  E – b undefined on line 4
  E – c undefined on lines 5 6
Execution suppressed due to compilation errors
%
```

Here we misspelled *output* and gave a FORTRAN style variable declaration which the translator diagnosed as a 'Malformed declaration'. When, on line 6, we used '(' and ')' for subscripting (as in FORTRAN) rather than the '[' and ']' which are used in Pascal, the translator noted that *a* was not defined as a **procedure**. This occurred because **procedure** and **function** argument lists are delimited by parentheses in Pascal. As it is not permissible to assign to procedure calls the translator diagnosed a malformed statement at the point of assignment.

Expected and unexpected end-of-file, "QUIT"

If the translator finds a complete program, but there is more non-comment text in the input file, then it 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. The message may appear after the recovery says that it "Expected `.'" since '.' is the symbol that terminates a program.

If severe errors in the input prohibit further processing the translator may produce a diagnostic followed by "QUIT". One example of this was given above – a non-terminated comment; another example is a line which is longer than 160 characters. Consider also the following example.

```
% pix -l mism.p
Berkeley Pascal PI -- Version 2.0 (Sat Oct 18 21:01:54 1980)

Tue Oct 14 21:38 1980  mism.p

    1  program mismatch(output)
    2  begin
e -----↑------ Inserted ';'
    3          writeln('***');
    4          { The next line is the last line in the file }
    5          writeln
E ---------------------↑------ Unexpected end-of-file – QUIT
%
```

3.2. Translator semantic errors

The extremely large number of semantic diagnostic messages which the translator produces make it unreasonable to discuss each message or group of messages in detail. The messages are, however, very informative. We will here explain the typical formats and the terminology used in the error messages so that you will be able to make sense out of them. In any case in which a diagnostic is not completely comprehensible you can refer to the *User Manual* by Jensen and Wirth for examples.

Format of the error diagnostics

As we saw in the example program above, the error diagnostics from the Pascal translator include the number of a line in the text of the program as well as the text of the error message. While this number is most often the line where the error occurred, it is occasionally the number of a line containing a bracketing keyword like **end** or **until**. In this case, the diagnostic may refer to the previous statement. This occurs because of the method the translator uses for sampling line numbers. The absence of a trailing ';' in the previous statement causes the line number corresponding to the **end** or **until**. to become associated with the statement. As Pascal is a free-format language, the line number associations can only be approximate and may seem arbitrary to some users. This is the only notable exception, however, to reasonable associations.

Incompatible types

Since Pascal is a strongly typed language, many semantic errors manifest themselves as type errors. These are called 'type clashes' by the translator. The types allowed for various operators in the language are summarized on page 108 of the Jensen-Wirth *User Manual*. It is important to know that the Pascal translator, in its diagnostics, distinguishes between the following type 'classes':

array	Boolean	char	file	integer
pointer	real	record	scalar	string

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:

```
Tue Oct 14 21:37 1980  clash.p:
E 7 – Type clash: integer is incompatible with char
  ... Type of expression clashed with type of variable in assignment
```

In this 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.

Scalar

The only class whose meaning is not self-explanatory is 'scalar'. Scalar has a precise meaning in the Jensen-Wirth *User Manual* where, in fact, it refers to *char*, *integer*, *real*, and *Boolean* types as well as the enumerated types. For the purposes of the Pascal translator, scalar in an error message refers to a user-defined, enumerated type, such as *ops* in the example above or *color* in

```
type color = (red, green, blue)
```

For integers, the more explicit denotation *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.

Function and procedure type errors

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:

```
Tue Oct 14 21:38 1980  sin1.p:
E 12 – sin takes exactly one argument
```

is given. If the type of the argument is wrong, a message like

```
Tue Oct 14 21:38 1980  sin2.p:
E 12 — sin's argument must be integer or real, not char
```

is produced. A few functions and procedures implemented in Pascal 6000-3.4 are diagnosed as unimplemented in Berkeley Pascal, notably those related to **segmented** files.

Can't read and write scalars, etc.

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)
```

"standard" Pascal does not associate these constants with the strings 'red', 'green', and 'blue' in any way. An extension has been added which allows enumerated types to be read and written, however if the program is to be portable, you will have to write your own routines to perform these functions. Standard Pascal only allows the reading of characters, integers and real numbers from text files. You cannot read strings or Booleans. It is possible to make a

```
file of color
```

but the representation is binary rather than string.

Expression diagnostics

The diagnostics for semantically ill-formed expressions are very explicit. Consider this sample translation:

```
% pi -l expr.p
Berkeley Pascal PI -- Version 2.0 (Sat Oct 18 21:01:54 1980)

Tue Oct 14 21:37 1980  expr.p

 1  program x(output);
 2  var
 3          a: set of char;
 4          b: Boolean;
 5          c: (red, green, blue);
 6          p: ↑ integer;
 7          A: alfa;
 8          B: packed array [1..5] of char;
 9  begin
10          b := true;
11          c := red;
12          new(p);
13          a := [];
14          A := 'Hello, yellow';
15          b := a and b;
16          a := a * 3;
```

```
   17        if input < 2 then writeln('boo');
   18        if p <= 2 then writeln('sure nuff');
   19        if A = B then writeln('same');
   20        if c = true then writeln('hue''s and color''s')
   21  end.
E 14 – Constant string too long
E 15 – Left operand of and must be Boolean, not set
E 16 – Cannot mix sets with integers and reals as operands of *
E 17 – files may not participate in comparisons
E 18 – pointers and integers cannot be compared – operator was <=
E 19 – Strings not same length in = comparison
E 20 – scalars and Booleans cannot be compared – operator was =
In program x:
  w – constant green is never used
  w – constant blue is never used
  w – variable B is used but never set
%
```

This example is admittedly far-fetched, but illustrates that the error messages are sufficiently clear to allow easy determination of the problem in the expressions.

Type equivalence

Several diagnostics produced 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 identifier. Thus, the variables x and y declared as

```
var
   x: ↑ integer;
   y: ↑ integer;
```

do not have the same type. The assignment

```
x := y
```

thus produces the diagnostics:

```
Tue Oct 14 21:38 1980  typequ.p:
E 7 – Type clash: non–identical pointer types
  ... Type of expression clashed with type of variable in assignment
```

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↑
```

is allowed in either case. 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.

Unreachable statements

Berkeley Pascal flags 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 diagnostic is produced for the statement:

```
if false then
    writeln('impossible!')
```

Goto's into structured statements

The translator detects and complains about **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 checks are made only within the body of a single procedure or function.

Unused variables, never set variables

Although *pi* always clears variables to 0 at **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 help detect errors in program logic, *pi* flags as a 'w' warning error:

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.

In fact, these diagnostics are applied 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.

3.3. Translator panics, i/o errors

Panics

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:

```
snark (rvalue) line=110 yyline=109
Snark in pi
```

If you receive such a message, the translation will be quickly and perhaps ungracefully terminated.

You should contact a teaching assistant or a member of the system staff, after saving a copy of your program for later inspection. If you were making changes to an existing program when the problem occurred, you may be able to work around the problem by ascertaining which change caused the *snark* and making a different change or correcting an error in the program. A small number of panics are possible in *px*. All panics should be reported to a teaching assistant or systems staff so that they can be fixed.

Out of memory

The only other error which will abort translation when no errors are detected is running out of memory. All tables in the translator, with the exception of the parse stack, are dynamically allocated, and can grow to take up the full available process space of 64000 bytes on the PDP-11. On the VAX-11, table sizes are extremely generous and very large (25000) line programs have been easily accommodated. For the PDP–11, it is generally true that the size of the largest translatable program is directly related to **procedure** and **function** size. A number of non-trivial Pascal programs, including some with more than 2000 lines and 2500 statements have been translated and interpreted using Berkeley Pascal on PDP-11's. Notable among these are the Pascal-S interpreter, a large set of programs for automated generation of code generators, and a general context-free parsing program which has been used to parse sentences with a grammar for a superset of English. In general, very large programs should be translated using *pc* and the separate compilation facility.

If you receive an 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 **function** into several routines.

I/O 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.

3.4. Run-time 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).

Start-up errors

These errors 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.

Program execution errors

These errors occur when the program interacts with the Pascal runtime environment in an inappropriate way. Typical errors are values or subscripts out of range, bad arguments to built-in functions, exceeding the statement limit because of an infinite loop, or running out of memory‡. The interpreter will produce a backtrace after the error occurs, showing all the active routine calls, unless the **p** option was disabled when the program was translated. Unfortunately, no variable values are given and no way of extracting them is available.*

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

‡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.)

program we get the following response.

```
% pix primes.p
Execution begins...
   2    3    5    7   11   13   17   19   23   29
  31   37   41   43   47   53   59   61   67   71
  73   79   83   89   97  101  103  107  109  113
 127  131  137  139  149  151  157  163  167

Subscript out of range

        Error in "primes"+8 near line 14.
Execution terminated abnormally.

941 statements executed in 0.50 seconds cpu time.
%
```

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.

Interrupts

If the 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*.

I/O interaction errors

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.

†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.

4. Input/output

This section describes features of the Pascal input/output environment, with special consideration of the features peculiar to an interactive implementation.

4.1. Introduction

Our first sample programs, in section 2, used 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 following Pascal program which is similar to the program *cat* (1).

```
% pix -l kat.p <primes
Berkeley Pascal PI -- Version 2.0 (Sat Oct 18 21:01:54 1980)

Tue Oct 14 21:38 1980  kat.p

    1  program kat(input, output);
    2  var
    3      ch: char;
    4  begin
    5      while not eof do begin
    6          while not eoln do begin
    7              read(ch);
    8              write(ch)
    9          end;
   10          readln;
   11          writeln
   12      end
   13  end { kat }.
Execution begins...
    2    3    5    7   11   13   17   19   23   29
   31   37   41   43   47   53   59   61   67   71
   73   79   83   89   97  101  103  107  109  113
  127  131  137  139  149  151  157  163  167  173
  179  181  191  193  197  199  211  223  227  229

Execution terminated.

925 statements executed in 0.15 seconds cpu time.
%
```

Here we have used the shell's syntax to redirect the program input from a file in *primes* in which we had placed the output of our 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 daemon *lpr* (1) before. Thus, the same output as above would be produced by

```
% cat primes | pix -l kat.p
```

All of these examples use the shell to control the input and output 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 created the file *data*. We then use it as input to another version of a listing program.

```
% cat data
line one.
line two.
line three is the end.
% pix -l copydata.p
Berkeley Pascal PI -- Version 2.0 (Sat Oct 18 21:01:54 1980)

Tue Oct 14 21:37 1980  copydata.p

    1  program copydata(data, output);
    2  var
    3      ch: char;
    4      data: text;
    5  begin
    6      reset(data);
    7      while not eof(data) do begin
    8          while not eoln(data) do begin
    9              read(data, ch);
   10              write(ch)
   11          end;
   12          readln(data);
   13          writeln
   14      end
   15  end { copydata }.
Execution begins...
line one.
line two.
line three is the end.
Execution terminated.

134 statements executed in 0.08 seconds cpu time.
%
```

By mentioning the file *data* in the **program** statement, we have indicated that we wish it to correspond to the UNIX file *data*. Then, when we 'reset(data)', the Pascal system opens our file 'data' for reading. More sophisticated, but less portable, examples of using UNIX files will be given in sections 4.5 and 4.6. There is a portability problem even with this simple example. Some Pascal systems attach meaning to the ordering of the file in the **program** statement file list. Berkeley Pascal does not do so.

4.2. Eof and eoln

An extremely common problem encountered by new users of Pascal, especially in the interactive environment offered by UNIX, relates to the definitions of *eof* and *eoln*. These functions are supposed to be defined at the beginning of execution of a Pascal program, indicating whether the input device is at the end of a line or the end of a file. Setting *eof* or *eoln* actually corresponds to an implicit read in which the input is inspected, but no input is "used up". In fact, there is no

way the system can know whether the input is at the end-of-file or the end-of-line unless it attempts to read a line from it. If the input is from a previously created file, then this reading can take place without run-time action by the user. However, if the input is from a terminal, then the input is what the user types.† If the system were to do an initial read automatically at the beginning of program execution, and if the input were a terminal, the user would have to type some input before execution could begin. This would make it impossible for the program to begin by prompting for input or printing a herald.

Berkeley Pascal has been designed so that an initial read is not necessary. At any given time, the Pascal system may or may not know whether the end-of-file or end-of-line conditions are true. Thus, internally, these functions can have three values – true, false, and "I don't know yet; if you ask me I'll have to find out". All files remain in this last, indeterminate state until the Pascal program requires a value for *eof* or *eoln* either explicitly or implicitly, e.g. in a call to *read*. The important point to note here is that if you force the Pascal system to determine whether the input is at the end-of-file or the end-of-line, it will be necessary for it to attempt to read from the input.

Thus consider the following example code

```
while not eof do begin
    write('number, please? ');
    read(i);
    writeln('that was a ', i: 2)
end
```

At first glance, this may be appear to be a correct program for requesting, reading and echoing numbers. Notice, however, that the **while** loop asks whether *eof* is true *before* the request is printed. This will force the Pascal system to decide whether the input is at the end-of-file. The Pascal system will give no messages; it will simply wait for the user to type a line. By producing the desired prompting before testing *eof*, the following code avoids this problem:

```
write('number, please ?');
while not eof do begin
    read(i);
    writeln('that was a ', i:2);
    write('number, please ?')
end
```

The user must still type a line before the **while** test is completed, but the prompt will ask for it. This example, however, is still not correct. To understand why, it is first necessary to know, as we will discuss below, that there is a blank character at the end of each line in a Pascal text file. The *read* procedure, when reading integers or real numbers, is defined so that, if there are only blanks left in the file, it will return a zero value and set the end-of-file condition. If, however, there is a number remaining in the file, the end-of-file condition will not be set even if it is the last number, as *read* never reads the blanks after the number, and there is always at least one blank. Thus the modified code will still put out a spurious

```
that was a 0
```

at the end of a session with it when the end-of-file is reached. The simplest way to correct the problem in this example is to use the procedure *readln* instead of *read* here. In general, unless we

†It is not possible to determine whether the input is a terminal, as the input may appear to be a file but actually be a *pipe*, the output of a program which is reading from the terminal.

test the end-of-file condition both before and after calls to *read* or *readln*, there will be inputs for which our program will attempt to read past end-of-file.

4.3. More about eoln

To have a good understanding of when *eoln* will be true it is necessary to know that in any file there is a special character indicating end-of-line, and that, in effect, the Pascal system always reads one character ahead of the Pascal *read* commands.† For instance, in response to 'read(ch)', the system sets *ch* to the current input character and gets the next input character. If the current input character is the last character of the line, then the next input character from the file is the new-line character, the normal UNIX line separator. When the read routine gets the new-line character, it replaces that character by a blank (causing every line to end with a blank) and sets *eoln* to true. *Eoln* will be true as soon as we read the last character of the line and **before** we read the blank character corresponding to the end of line. Thus it is almost always a mistake to write a program which deals with input in the following way:

```
read(ch);
if eoln then
    Done with line
else
    Normal processing
```

as this will almost surely have the effect of ignoring the last character in the line. The 'read(ch)' belongs as part of the normal processing.

Given this framework, it is not hard to explain the function of a *readln* call, which is defined as:

```
while not eoln do
    get(input);
get(input);
```

This advances the file until the blank corresponding to the end-of-line is the current input symbol and then discards this blank. The next character available from *read* will therefore be the first character of the next line, if one exists.

4.4. Output buffering

A final point about Pascal input-output must be noted here. This concerns the buffering of the file *output*. It is extremely inefficient for the Pascal system to send each character to the user's terminal as the program generates it for output; even less efficient if the output is the input of another program such as the line printer daemon *lpr* (1). To gain efficiency, the Pascal system "buffers" the output characters (i.e. it saves them in memory until the buffer is full and then emits the entire buffer in one system interaction.) However, to allow interactive prompting to work as in the example given above, this prompt must be printed before the Pascal system waits for a response. For this reason, Pascal normally prints all the output which has been generated for the file *output* whenever

1) A *writeln* occurs, or
2) The program reads from the terminal, or
3) The procedure *message* or *flush* is called.

---

†In Pascal terms, 'read(ch)' corresponds to 'ch := input^; get(input)'

Thus, in the code sequence

```
for i := 1 to 5 do begin
   write(i: 2);
   Compute a lot with no output
end;
writeln
```

the output integers will not print until the *writeln* occurs. The delay can be somewhat disconcerting, and you should be aware that it will occur. By setting the **b** option to 0 before the **program** statement by inserting a comment of the form

```
(*$b0*)
```

we can cause *output* to be completely unbuffered, with a corresponding horrendous degradation in program efficiency. Option control in comments is discussed in section 5.

4.5. Files, reset, and rewrite

It is possible to use extended forms of the built-in functions *reset* and *rewrite* to get more general associations of UNIX file names with Pascal file variables. When a file other than *input* or *output* is to be read or written, then the reading or writing must be preceded by a *reset* or *rewrite* call. In general, if the Pascal file variable has never been used before, there will be no UNIX filename associated with it. As we saw in section 2.9, by mentioning the file in the **program** statement, we could cause a UNIX file with the same name as the Pascal variable to be associated with it. If we do not mention a file in the **program** statement and use it for the first time with the statement

```
reset(f)
```

or

```
rewrite(f)
```

then the Pascal system will generate a temporary name of the form 'tmp.x' for some character 'x', and associate this UNIX file name name with the Pascal file. The first such generated name will be 'tmp.1' and the names continue by incrementing their last character through the ASCII set. The advantage of using such temporary files is that they are automatically *remove*d by the Pascal system as soon as they become inaccessible. They are not removed, however, if a runtime error causes termination while they are in scope.

To cause a particular UNIX pathname to be associated with a Pascal file variable we can give that name in the *reset* or *rewrite* call, e.g. we could have associated the Pascal file *data* with the file 'primes' in our example in section 3.1 by doing:

```
reset(data, 'primes')
```

instead of a simple

```
reset(data)
```

In this case it is not essential to mention 'data' in the program statement, but it is still a good idea because is serves as an aid to program documentation. The second parameter to *reset* and

rewrite may be any string value, including a variable. Thus the names of UNIX files to be associated with Pascal file variables can be read in at run time. Full details on file name/file variable associations are given in section A.3.

4.6. Argc and argv

Each UNIX process receives a variable length sequence of arguments each of which is a variable length character string. The built-in function *argc* and the built-in procedure *argv* can be used to access and process these arguments. The value of the function *argc* is the number of arguments to the process. By convention, the arguments are treated as an array, and indexed from 0 to *argc*−1, with the zeroth argument being the name of the program being executed. The rest of the arguments are those passed to the command on the command line. Thus, the command

```
% obj /etc/motd /usr/dict/words hello
```

will invoke the program in the file *obj* with *argc* having a value of 4. The zeroth element accessed by *argv* will be 'obj', the first '/etc/motd', etc.

Pascal does not provide variable size arrays, nor does it allow character strings of varying length. For this reason, *argv* is a procedure and has the syntax

```
argv(i, a)
```

where *i* is an integer and *a* is a string variable. This procedure call assigns the (possibly truncated or blank padded) *i* 'th argument of the current process to the string variable *a*. The file manipulation routines *reset* and *rewrite* will strip trailing blanks from their optional second arguments so that this blank padding is not a problem in the usual case where the arguments are file names.

We are now ready to give a Berkeley Pascal program 'kat', based on that given in section 3.1 above, which can be used with the same syntax as the UNIX system program *cat* (1).

```
% cat kat.p
program kat(input, output);
var
    ch: char;
    i: integer;
    name: packed array [1..100] of char;
begin
    i := 1;
    repeat
        if i < argc then begin
            argv(i, name);
            reset(input, name);
            i := i + 1
        end;
        while not eof do begin
            while not eoln do begin
                read(ch);
                write(ch)
            end;
            readln;
            writeln
```

```
      end
    until i >= argc
end { kat }.
%
```

Note that the *reset* call to the file *input* here, which is necessary for a clear program, may be disallowed on other systems. As this program deals mostly with *argc* and *argv* and UNIX system dependent considerations, portability is of little concern.

If this program is in the file 'kat.p', then we can do

```
% pi kat.p
% mv obj kat
% kat primes
    2    3    5    7   11   13   17   19   23   29
   31   37   41   43   47   53   59   61   67   71
   73   79   83   89   97  101  103  107  109  113
  127  131  137  139  149  151  157  163  167  173
  179  181  191  193  197  199  211  223  227  229


930 statements executed in 0.18 seconds cpu time.
% kat
This is a line of text.
This is a line of text.
The next line contains only an end-of-file (an invisible control-d!)
The next line contains only an end-of-file (an invisible control-d!)

287 statements executed in 0.03 seconds cpu time.
%
```

Thus we see that, if it is given arguments, 'kat' will, like *cat,* copy each one in turn. If no arguments are given, it copies from the standard input. Thus it will work as it did before, with

```
% kat < primes
```

now equivalent to

```
% kat primes
```

although the mechanisms are quite different in the two cases. Note that if 'kat' is given a bad file name, for example:

```
% kat xxxxqqq


Could not open xxxxqqq: No such file or directory

    Error in "kat"+5 near line 11.

4 statements executed in 0.02 seconds cpu time.
```

```
%
```

it will give a diagnostic and a post-mortem control flow backtrace for debugging. If we were going to use 'kat', we might want to translate it differently, e.g.:

```
% pi -pb kat.p
% mv obj kat
```

Here we have disabled the post-mortem statistics printing, so as not to get the statistics or the full traceback on error. The **b** option will cause the system to block buffer the input/output so that the program will run more efficiently on large files. We could have also specified the **t** option to turn off runtime tests if that was felt to be a speed hindrance to the program. Thus we can try the last examples again:

```
% kat xxxxqqq


Could not open xxxxqqq: No such file or directory

    Error in "kat"
% kat primes
  2   3   5   7  11  13  17  19  23  29
 31  37  41  43  47  53  59  61  67  71
 73  79  83  89  97 101 103 107 109 113
127 131 137 139 149 151 157 163 167 173
179 181 191 193 197 199 211 223 227 229

%
```

The interested reader may wish to try writing a program which accepts command line arguments like *pi* does, using *argc* and *argv* to process them.

5. Details on the components of the system

5.1. Options

The programs *pi*, *pc*, and *pxp* take a number of options.† There is a standard UNIX convention for passing options to programs on the command line, and this convention is followed by the Berkeley Pascal system programs. As we saw in the examples above, option related arguments consisted of the character '–' followed by a single character option name.

Except for the **b** option which takes a single digit value, each option may be set on (enabled) or off (disabled.) When an on/off valued option appears on the command line of *pi* or it inverts the default setting of that option. Thus

```
% pi -l foo.p
```

enables the listing option **l**, since it defaults off, while

```
% pi -t foo.p
```

disables the run time tests option **t**, since it defaults on.

In additon to inverting the default settings of *pi* options on the command line, it is also possible to control the *pi* options within the body of the program by using comments of a special form illustrated by

```
{$l-}
```

Here we see that the opening comment delimiter (which could also be a '(*') is immediately followed by the character '$'. After this '$', which signals the start of the option list, we can place a sequence of letters and option controls, separated by ',' characters‡. The most basic actions for options are to set them, thus

```
{$l+ Enable listing}
```

or to clear them

```
{$t-,p- No run-time tests, no post mortem analysis}
```

Notice that '+' always enables an option and '–' always disables it, no matter what the default is. Thus '–' has a different meaning in an option comment than it has on the command line. As shown in the examples, normal comment text may follow the option list.

5.2. Options common to Pi, Pc, and Pix

The following options are common to both the compiler and the interpreter. With each option we give its default setting, the setting it would have if it appeared on the command line, and a sample command using the option. Most options are on/off valued, with the **b** option taking a single digit value.

Buffering of the file output – b

The **b** option controls the buffering of the file *output*. The default is line buffering, with flushing at each reference to the file *input* and under certain other circumstances detailed in section

†As *piz* uses *pi* to translate Pascal programs, it takes the options of *pi* also. We refer to them here, however, as *pi* options.

‡This format was chosen because it is used by Pascal 6000-3.4. In general the options common to both implementations are controlled in the same way so that comment control in options is mostly portable. It is recommended, however, that only one control be put per comment for maximum portability, as the Pascal 6000-3.4 implementation will ignore controls after the first one which it does not recognize.

5 below. Mentioning **b** on the command line, e.g.

```
% pi -b assembler.p
```

causes standard output to be block buffered, where a block is some system-defined number of characters. The **b** option may also be controlled in comments. It, unique among the Berkeley Pascal options, takes a single digit value rather than an on or off setting. A value of 0, e.g.

```
{$b0}
```

causes the file *output* to be unbuffered. Any value 2 or greater causes block buffering and is equivalent to the flag on the command line. The option control comment setting **b** must precede the **program** statement.

Include file listing – i

The **i** option takes the name of an **include** file, **procedure** or **function** name and causes it to be listed while translating†. Typical uses would be

```
% pix -i scanner.i compiler.p
```

to make a listing of the routines in the file scanner.i, and

```
% pix -i scanner compiler.p
```

to make a listing of only the routine *scanner*. This option is especially useful for conservation-minded programmers making partial program listings.

Make a listing – l

The **l** option enables a listing of the program. The **l** option defaults off. When specified on the command line, it causes a header line identifying the version of the translator in use and a line giving the modification time of the file being translated to appear before the actual program listing. The **l** option is pushed and popped by the **i** option at appropriate points in the program.

Standard Pascal only – s

The **s** option causes many of the features of the UNIX implementation which are not found in standard Pascal to be diagnosed as 's' warning errors. This option defaults off and is enabled when mentioned on the command line. Some of the features which are diagnosed are: non-standard **procedure**s and **function**s, extensions to the **procedure** *write*, and the padding of constant strings with blanks. In addition, all letters are mapped to lower case except in strings and characters so that the case of keywords and identifiers is effectively ignored. The **s** option is most useful when a program is to be transported, thus

```
% pi -s isitstd.p
```

will produce warnings unless the program meets the standard.

Runtime tests – t and C

These options control the generation of tests that subrange variable values are within bounds at run time. *pi* defaults to generating tests and uses the option **t** to disable them. *pc* defaults to not generating tests, and uses the option **C** to enable them. Disabling runtime tests also causes

†Include files are discussed in section 5.9.

assert statements to be treated as comments.‡

Suppress warning diagnostics – w

The **w** option, which defaults on, allows the translator to print a number of warnings about inconsistencies it finds in the input program. Turning this option off with a comment of the form

```
{$w-}
```

or on the command line

```
% pi -w tryme.p
```

suppresses these usually useful diagnostics.

Generate counters for a pxp execution profile – z

The **z** option, which defaults off, enables the production of execution profiles. By specifying **z** on the command line, i.e.

```
% pi -z foo.p
```

or by enabling it in a comment before the **program** statement causes *pi* and *pc* to insert operations in the interpreter code to count the number of times each statement was executed. An example of using *pxp* was given in section 2.6; its options are described in section 5.6. Note that the **z** option cannot be used on separately compiled programs.

5.3. Options available in Pi

Post-mortem dump – p

The **p** option defaults on, and causes the runtime system to initiate a post-mortem backtrace when an error occurs. It also cause *px* to count statements in the executing program, enforcing a statement limit to prevent infinite loops. Specifying **p** on the command line disables these checks and the ability to give this post-mortem analysis. It does make smaller and faster programs, however. It is also possible to control the **p** option in comments. To prevent the post-mortem backtrace on error, **p** must be off at the end of the **program** statement. Thus, the Pascal cross-reference program was translated with

```
% pi -pbt pxref.p
```

5.4. Options available in Px

The first argument to *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 avail-

‡See section A.1 for a description of **assert** statements.

able to the Pascal program by using the built-ins *argc* and *argv* as described in section 4.6.

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
```

creates a file *foo.s*. No executable file is created.

Symbolic Debugger Information – g

The **g** option causes the compiler to generate information needed by *sdb*(1) the symbolic debugger. For a complete description of *sdb* see Volume 2c of the UNIX Reference Manual.

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
```

causes the compiled program to be called *myprog*.

Generate counters for a *prof* execution profile – p

The compiler produces code which counts the number of times each routine is called. The profiling is based on a periodic sample taken by the system rather than by inline counters used by *pxp*. This results in less degradation in execution, at somewhat of a loss in accuracy. See *prof*(1) for a more complete description.

Run the object code optimizer – O

The output of the compiler is run through the object code optimizer. This provides an increase in compile time in exchange for a decrease in compiled code size and execution time.

5.6. Options available in Pxp

Pxp takes, on its command line, a list of options followed by 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 none of these options is specified, then *pxp* functions as a program reformatter.

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.

The following options are relevant to profiling with *pxp*:

Include the bodies of all routines in the profile – a

Pxp normally suppresses printing the bodies of routines which were never executed, to make the profile more compact. This option forces all routine bodies to be printed.

Suppress declaration parts from a profile – d

Normally a profile includes declaration parts. Specifying **d** on the command line suppresses declaration parts.

Eliminate include directives – e

Normally, *pxp* preserves **include** directives to the output when reformatting a program, as though they were comments. Specifying **–e** causes the contents of the specified files to be reformatted into the output stream instead. This is an easy way to eliminate **include** directives, e.g. before transporting a program.

Fully parenthesize expressions – f

Normally *pxp* prints expressions with the minimal parenthesization necessary to preserve the structure of the input. This option causes *pxp* to fully parenthesize expressions. Thus the statement which prints as

```
d := a + b mod c / e
```

with minimal parenthesization, the default, will print as

```
d := a + ((b mod c) / e)
```

with the **f** option specified on the command line.

Left justify all procedures and functions – j

Normally, each **procedure** and **function** body is indented to reflect its static nesting depth. This option prevents this nesting and can be used if the indented output would be too wide.

Print a table summarizing procedure and function calls – t

The **t** option causes *pxp* to print a table summarizing the number of calls to each **procedure** and **function** in the program. It may be specified in combination with the **z** option, or separately.

Enable and control the profile – z

The **z** profile option is very similar to the **i** listing control option of *pi*. If **z** is specified on the command line, then all arguments up to the source file argument which ends in '.p' are taken to be the names of **procedures** and **functions** or **include** files which are to be profiled. If this list is null, then the whole file is to be profiled. A typical command for extracting a profile of part of a large program would be

```
% pxp -z test parser.i compiler.p
```

This specifies that profiles of the routines in the file *parser.i* and the routine *test* are to be made.

5.7. Formatting programs using 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.

Pxp automatically paragraphs the program, performing housekeeping chores such as comment alignment, and treating blank lines, lines containing exactly one blank and lines containing only a form-feed character as though they were comments, preserving their vertical spacing effect in the output. *Pxp* distinguishes between four kinds of comments:

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 the 40th column from the current "left margin".

Consider the following program.

```
% cat comments.p
{ This is a left marginal comment. }
program hello(output);
var i : integer; {This is a trailing comment}
j : integer;    {This is a right marginal comment}
k : array [ 1..10] of array [1..10] of integer; {Marginal, but past the margin}
{
 An aligned, multi-line comment
 which explains what this program is
 all about
}
begin
i := 1; {Trailing i comment}
{A left marginal comment}
 {An aligned comment}
j := 1;         {Right marginal comment}
k[1] := 1;
writeln(i, j, k[1])
end.
```

When formatted by *pxp* the following output is produced.

```
% pxp comments.p
{ This is a left marginal comment. }

program hello(output);
var
   i: integer; {This is a trailing comment}
   j: integer;                                    {This is a right marginal comment}
   k: array [1..10] of array [1..10] of integer;  {Marginal, but past the margin}
{
 An aligned, multi-line comment
 which explains what this program is
 all about
}
begin
   i := 1; {Trailing i comment}
{A left marginal comment}
   {An aligned comment}
   j := 1;                                        {Right marginal comment}
   k[1] := 1;
   writeln(i, j, k[1])
end.
%
```

The following formatting related options are currently available in *pxp*. The options **f** and **j** described in the previous section may also be of interest.

Strip comments –s

The **s** option causes *pxp* to remove all comments from the input text.

Underline keywords – _

A command line argument of the form –_ as in

```
% pxp -_ dirty.p
```

can be used to cause *pxp* to underline all keywords in the output for enhanced readability.

Specify indenting unit – [23456789]

The normal unit which *pxp* uses to indent a structure statement level is 4 spaces. By giving an argument of the form $-d$ with d a digit, $2 \leq d \leq 9$ you can specify that d spaces are to be used per level instead.

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
```

The cross-reference is, unfortunately, not block structured. Full details on *pxref* are given in its

manual section *pxref* (1).

5.9. Multi-file programs

A text inclusion facility is available with Berkeley Pascal. 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 in editing, listing, and maintenance.

The **include** facility is based on that of the UNIX C compiler. To trigger it you can place the character '#' in the first portion of a line and then, after an arbitrary number of blanks or tabs, the word 'include' followed by a filename enclosed in single ' ' or double " " quotation marks. The file name may be followed by a semicolon ';' if you wish to treat this as a pseudo-Pascal statement. The filenames of included files must end in '.i'. An example of the use of included files in a main program would be:

```
program compiler(input, output, obj);

#include "globals.i"
#include "scanner.i"
#include "parser.i"
#include "semantics.i"

begin
   { main program }
end.
```

At the point the **include** pseudo-statement is encountered in the input, the lines from the included file are interpolated into the input stream. For the purposes of translation and runtime diagnostics and statement numbers in the listings and post-mortem backtraces, the lines in the included file are numbered from 1. Nested includes are possible up to 10 deep.

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.

For the purposes of error diagnostics when not making a listing, the filename will be printed before each diagnostic if the current filename has changed since the last filename was printed.

5.10. Separate Compilation with Pc

A separate compilation facility is provided with the Berkeley Pascal compiler, *pc*. This facility allows programs to be divided into a number of files and the pieces to be compiled individually, to be linked together at some later time. This is especially useful for large programs, where small changes would otherwise require time-consuming re-compilation of the entire program.

Normally, *pc* expects to be given entire Pascal programs. However, if given the **–c** option on the command line, it will accept a sequence of definitions and declarations, and compile them into a **.o** file, to be linked with a Pascal program at a later time. In order that procedures and functions be available across separately compiled files, they must be declared with the directive **external**. This directive is similar to the directive **forward** in that it must precede the resolution of the function or procedure, and formal parameters and function result types must be specified at the **external** declaration and may not be specified at the resolution.

Type checking is performed across separately compiled files. Since Pascal type defintions define unique types, any types which are shared between separately compiled files must be the same definition. This seemingly impossible problem is solved using a facility similar to the **include** facility discussed above. Definitions may be placed in files with the extension **.h** and the files included by separately compiled files. Each definition from a **.h** file defines a unique type, and all uses of a definition from the same **.h** file define the same type. Similarly, the facility is extended to allow the definition of **const**s and the declaration of **label**s, **var**s, and **external function**s and **procedure**s. Thus **procedure**s and **function**s which are used between separately compiled files must be declared **external**, and must be so declared in a **.h** file included by any file which calls or resolves the **function** or **procedure**. Conversely, **function**s and **procedure**s declared **external** may only be so declared in **.h** files. These files may be included only at the outermost level, and thus define or declare global objects. Note that since only **external function** and **procedure** declarations (and not resolutions) are allowed in **.h** files, statically nested **function**s and **procedure**s can not be declared **external**.

An example of the use of included **.h** files in a program would be:

```
program compiler(input, output, obj);

#include "globals.h"
#include "scanner.h"
#include "parser.h"
#include "semantics.h"

begin
   { main program }
end.
```

This might include in the main program the definitions and declarations of all the global **label**s, **const**s, **type**s **var**s from the file globals.h, and the **external function** and **procedure** declarations for each of the separately compiled files for the scanner, parser and semantics. The header file *scanner.h* would contain declarations of the form:

```
type
   token = record
      { token fields }
   end;

function scan(var inputfile: text): token;
   external;
```

Then the scanner might be in a separately compiled file containing:

```
#include "globals.h"
#include "scanner.h"

function scan;
begin
   { scanner code }
end;
```

which includes the same global definitions and declarations and resolves the scanner functions and procedures declared **external** in the file scanner.h.

A. Appendix to Wirth's Pascal Report

This section is an appendix to the definition of the Pascal language in Niklaus Wirth's *Pascal Report* and, with that Report, precisely defines the Berkeley implementation. This appendix includes a summary of extensions to the language, gives the ways in which the undefined specifications were resolved, gives limitations and restrictions of the current implementation, and lists the added functions and procedures available. It concludes with a list of differences with the commonly available Pascal 6000–3.4 implementation, and some comments on standard and portable Pascal.

A.1. Extensions to the language Pascal

This section defines non-standard language constructs available in Berkeley Pascal. The **s** standard Pascal option of the translators *pi* and *pc* can be used to detect these extensions in programs which are to be transported.

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 x(output);
var z : packed array [ 1 .. 13 ] of char;
begin
   z := 'red';
   writeln(z)
end;
```

The padded blanks are added on the right. Thus the assignment above is equivalent to:

```
z := 'red          '
```

which is standard Pascal.

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)
```

cause the internal representation of expression *a*, which must be Boolean, character, integer, pointer, or a user-defined enumerated type, to be written in octal or hexadecimal respectively.

Assert statement

An **assert** statement causes a *Boolean* expression to be evaluated each time the statement is executed. A runtime error results if any of the expressions evaluates to be *false*. The **assert** statement is treated as a comment if run-time tests are disabled. The syntax for **assert** is:

assert <expr>

Enumerated type input-output

Enumerated types may be read and written. On output the string name associated with the enumerated value is output. If the value is out of range, a runtime error occurs. On input an identifier is read and looked up in a table of names associated with the type of the variable, and the appropriate internal value is assigned to the variable being read. If the name is not found in the table a runtime error occurs.

Structure returning functions

An extension has been added which allows functions to return arbitrary sized structures rather than just scalars as in the standard.

Separate compilation

The compiler *pc* has been extended to allow separate compilation of programs. Procedures and functions declared at the global level may be compiled separately. Type checking of calls to separately compiled routines is performed at load time to insure that the program as a whole is consistent. See section 5.10 for details.

A.2. Resolution of the undefined specifications

File name – file variable associations

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.

3) If a file which has never had UNIX name associated is reset or rewritten without specifying a name via the second argument, then a temporary name of the form 'tmp.x' is associated with the file. Temporary names start with 'tmp.1' and continue by incrementing the last character in the USASCII ordering. Temporary files are removed automatically when their scope is exited.

The program statement

The syntax of the **program** statement is:

program <id> (<file id> { , <file id > }) ;

The file identifiers (other than *input* and *output*) must be declared as variables of **file** type in the global declaration part.

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) The procedure *reset* may be used on *input*. If no UNIX file name has ever been associated with *input*, and no file name is given, then an attempt will be made to 'rewind' *input*. If this fails, a run time error will occur. *Rewrite* calls to output act as for any other file, except that *output* initially has no associated file. This means that a simple

```
rewrite(output)
```

associates a temporary name with *output*.

Details for files

If a file other than *input* is to be read, then reading must be initiated by a call to the procedure *reset* which causes the Pascal system to attempt to open the associated UNIX file for reading. If this fails, then a runtime error occurs. Writing of a file other than *output* must be initiated by a *rewrite* call, which causes the Pascal system to create the associated UNIX file and to then open the file for writing only.

Buffering

The buffering for *output* is determined by the value of the **b** option at the end of the **program** statement. If it has its default value 1, then *output* is buffered in blocks of up to 512 characters, flushed whenever a writeln occurs and at each reference to the file *input*. If it has the value 0, *output* is unbuffered. Any value of 2 or more gives block buffering without line or *input* reference flushing. All other output files are always buffered in blocks of 512 characters. All output buffers are flushed when the files are closed at scope exit, whenever the procedure *message* is called, and can be flushed using the built-in procedure *flush*.

An important point for an interactive implementation is the definition of 'input↑'. If *input* is a teletype, and the Pascal system reads a character at the beginning of execution to define 'input↑', then no prompt could be printed by the program before the user is required to type some input. For this reason, 'input↑' is not defined by the system until its definition is needed, reading from a file occurring only when necessary.

The character set

Seven bit USASCII is the character set used on UNIX. The standard Pascal symbols 'and', 'or', 'not', '<=', '>=', '<>', and the uparrow '↑' (for pointer qualification) are recognized.† Less portable are the synonyms tilde '~' for **not**, '&' for **and**, and '|' for **or**.

Upper and lower case are considered to be distinct. Keywords and built-in **procedure** and **function** names are composed of all lower case letters. Thus the identifiers GOTO and GOto are distinct both from each other and from the keyword **goto**. The standard type 'boolean' is also

†On many terminals and printers, the up arrow is represented as a circumflex '^'. These are not distinct characters, but rather different graphic representations of the same internal codes.
The proposed standard for Pascal considers them to be the same.

available as 'Boolean'.

Character strings and constants may be delimited by the character ' or by the character '#'; the latter is sometimes convenient when programs are to be transported. Note that the '#' character has special meaning when it is the first character on a line – see *Multi-file programs* below.

The standard types

The standard type *integer* is conceptually defined as

```
type integer = minint .. maxint;
```

Integer is implemented with 32 bit twos complement arithmetic. Predefined constants of type *integer* are:

```
const maxint = 2147483647; minint = −2147483648;
```

The standard type *char* is conceptually defined as

```
type char = minchar .. maxchar;
```

Built-in character constants are 'minchar' and 'maxchar', 'bell' and 'tab'; ord(minchar) = 0, ord(maxchar) = 127.

The type *real* is implemented using 64 bit floating point arithmetic. The floating point arithmetic is done in 'rounded' mode, and provides approximately 17 digits of precision with numbers as small as 10 to the negative 38th power and as large as 10 to the 38th power.

Comments

Comments can be delimited by either '{' and '}' or by '(*' and '*)'. If the character '{' appears in a comment delimited by '{' and '}', a warning diagnostic is printed. A similar warning will be printed if the sequence '(*' appears in a comment delimited by '(*' and '*)'. The restriction implied by this warning is not part of standard Pascal, but detects many otherwise subtle errors.

Option control

Options of the translators may be controlled in two distinct ways. A number of options may appear on the command line invoking the translator. These options are given as one or more strings of letters preceded by the character '–' and cause the default setting of each given option to be changed. This method of communication of options is expected to predominate for UNIX. Thus the command

```
% pi –l –s foo.p
```

translates the file foo.p with the listing option enabled (as it normally is off), and with only standard Pascal features available.

If more control over the portions of the program where options are enabled is required, then option control in comments can and should be used. The format for option control in comments is identical to that used in Pascal 6000–3.4. One places the character '$' as the first character of the comment and follows it by a comma separated list of directives. Thus an equivalent to the command line example given above would be:

{$l+,s+ listing on, standard Pascal}

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.

Directives consist of a letter designating the option, followed either by a '+' to turn the option on, or by a '–' to turn the option off. The **b** option takes a single digit instead of a '+' or '–'.

Notes on the listings

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.

Within the body of the listing, lines are numbered consecutively and correspond to the line numbers for the editor. Currently, two special kinds of lines may be used to format the listing: a line consisting of a form-feed character, control-l, which causes a page eject in the listing, and a line with no characters which causes the line number to be suppressed in the listing, creating a truly blank line. These lines thus correspond to 'eject' and 'space' macros found in many assemblers. Non-printing characters are printed as the character '?' in the listing.†

The standard procedure write

If no minimum field length parameter is specified for a *write*, the following default values are assumed:

integer	10
real	22
Boolean	length of 'true' or 'false'
char	1
string	length of the string
oct	11
hex	8

The end of each line in a text file should be explicitly indicated by 'writeln(f)', where 'writeln(output)' may be written simply as 'writeln'. For UNIX, the built-in function 'page(f)' puts a single ASCII form-feed character on the output file. For programs which are to be transported the filter *pcc* can be used to interpret carriage control, as UNIX does not normally do so.

A.3. Restrictions and limitations

Files

Files cannot be members of files or members of dynamically allocated structures.

Arrays, sets and strings

The calculations involving array subscripts and set elements are done with 16 bit arithmetic. This restricts the types over which arrays and sets may be defined. The lower bound of such a range must be greater than or equal to –32768, and the upper bound less than 32768. In particular, strings may have any length from 1 to 65535 characters, and sets may contain no more than 65535 elements.

---

†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.

Line and symbol length

There is no intrinsic limit on the length of identifiers. 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 however, currently exceeding 160 characters.

Procedure and function nesting and program size

At most 20 levels of **procedure** and **function** nesting are allowed. There is no fundamental, translator defined limit on the size of the program which can be translated. The ultimate limit is supplied by the hardware and thus, on the PDP-11, by the 16 bit address space. If one runs up against the 'ran out of memory' diagnostic the program may yet translate if smaller procedures are used, as a lot of space is freed by the translator at the completion of each **procedure** or **function** in the current implementation.

On the VAX-11, there is an implementation defined limit of 65536 bytes per variable. There is no limit on the number of variables.

Overflow

There is currently no checking for overflow on arithmetic operations at run-time on the PDP-11. Overflow checking is performed on the VAX-11 by the hardware.

A.4. Added types, operators, procedures and functions

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

In most cases the context of an expression involving a constant set allows the translator to determine the type of the set, even though the constant set itself may not uniquely determine this type. In the cases where it is not possible to determine the type of the set from local context, the expression type defaults to a set over the entire base type unless the base type is integer†. In the latter case the type defaults to the current binding of *intset,* which must be "type set of (a subrange of) integer" at that point.

Note that if *intset* is redefined via:

type intset = **set of** 0..58;

then the default integer set is the implicit *intset* of Pascal 6000–3.4

Additional predefined operators

The relationals '<' and '>' of proper set inclusion are available. With a and b sets, note that

(not $(a < b)$**)** $<>$ $(a >= b)$

†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.

As an example consider the sets a = [0,2] and b = [1]. The only relation true between these sets is '<>'.

Non-standard 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.
time(a)	causes the current time in the form ' hh:mm:ss ' to be assigned to the alfa variable *a*.

Non-standard 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*))).

‡Currently ignored by pdp-11 *px*.

random(x)	where *x* is a real parameter, evaluated but otherwise ignored, invokes a linear congruential random number generator. Successive seeds are generated as (seed*a + c) mod m and the new random number is a normalization of the seed to the range 0.0 to 1.0; a is 62605, c is 113218009, and m is 536870912. The initial seed is 7774755.
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.
wallclock	an integer function of no arguments returns the time in seconds since 00:00:00 GMT January 1, 1970.

A.5. Remarks on standard and portable Pascal

It is occasionally desirable to prepare Pascal programs which will be acceptable at other Pascal installations. While certain system dependencies are bound to creep in, judicious design and programming practice can usually eliminate most of the non-portable usages. Wirth's *Pascal Report* concludes with a standard for implementation and program exchange.

In particular, the following 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 not available in Berkeley Pascal

Segmented files and associated functions and procedures.

The function *trunc* with two arguments.

Arrays whose indices exceed the capacity of 16 bit arithmetic.

Features available in Berkeley Pascal but not in Pascal 6000-3.4

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.

Comparison of records.

---

†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**.

Other problem areas

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.

The 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 with them.

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.

For 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 the William Joy by an IBM Graduate Fellowship are gratefully acknowledged.

Berkeley Pascal PX Implementation Notes
Version 2.0 – January, 1979

William N. Joy[†]

M. Kirk McKusick[‡]

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

ABSTRACT

Berkeley Pascal is designed for interactive instructional use and runs on the VAX 11/780. The interpreter *px* executes the Pascal binaries generated by the Pascal translator *pi*.

The *PX Implementation Notes* describe the general organization of *px*, detail the various operations of the interpreter, and describe the file input/output structure. Conclusions are given on the viability of an interpreter based approach to language implementation for an instructional environment.

September 16, 1986

Berkeley Pascal PX Implementation Notes
Version 2.0 – January, 1979

William N. Joy†

M. Kirk McKusick‡

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

Introduction

These *PX Implementation Notes* have been updated from the original PDP 11/70 implementation notes to reflect the interpreter that runs on the VAX 11/780. These notes consist of four major parts. The first part outlines the general organization of *px*. Section 2 describes the operations (instructions) of the interpreter while section 3 focuses on input/output related activity. A final section gives conclusions about the viability of an interpreter based approach to language implementation for instruction.

Related Berkeley Pascal documents

The *PXP Implementation Notes* give details of the internals of the execution profiler *pxp;* parts of the interpreter related to *pxp* are discussed in section 2.10. A paper describing the syntactic error recovery mechanism used in *pi* was presented at the ACM Conference on Compiler Construction in Boulder Colorado in August, 1979.

Acknowledgements

This version of *px* is a PDP 11/70 to VAX 11/780 opcode mapping of the original *px* that was designed and implemented by Ken Thompson, with extensive modifications and additions by William Joy and Charles Haley. Without their work, this Berkeley Pascal system would never have existed. These notes were first written by William Joy for the PDP 11/70 implementation. We would also like to thank our faculty advisor Susan L. Graham for her encouragement, her helpful comments and suggestions relating to Berkeley Pascal and her excellent editorial assistance.

---

† The financial support of the National Science Foundation under grants MCS74-07644-A03 and MCS78-07291 and of an IBM Graduate Fellowship are gratefully acknowledged.

‡ The financial support of a Howard Hughes Graduate Fellowship is gratefully acknowledged.

1. Organization

Most of *px* is written in the VAX 11/780 assembly language, using the UNIX† assembler *as*. Portions of *px* are also written in the UNIX systems programming language C. *Px* consists of a main procedure that reads in the interpreter code, a main interpreter loop that transfers successively to various code segments implementing the abstract machine operations, built-in procedures and functions, and several routines that support the implementation of the Pascal input-output environment.

The interpreter runs at a fraction of the speed of equivalent compiled C code, with this fraction varying from 1/5 to 1/15. The interpreter occupies 18.5K bytes of instruction space, shared among all processes executing Pascal, and has 4.6K bytes of data space (constants, error messages, etc.) a copy of which is allocated to each executing process.

1.1. Format of the object file

Px normally interprets the code left in an object file by a run of the Pascal translator *pi*. The file where the translator puts the object originally, and the most commonly interpreted file, is called *obj*. In order that all persons using *px* share a common text image, this executable file is a small process that coordinates with the interpreter to start execution. The interpreter code is placed at the end of a special "header" file and the size of the initialized data area of this header file is expanded to include this code, so that during execution it is located at an easily determined address in its data space. When executed, the object process creates a *pipe*, creates another process by doing a *fork*, and arranges that the resulting parent process becomes an instance of *px*. The child process then writes the interpreter code through the pipe that it has to the interpreter process parent. When this process is complete, the child exits.

The real advantage of this approach is that it does not require modifications to the shell, and that the resultant objects are "true objects" not requiring special treatment. A simpler mechanism would be to determine the name of the file that was executed and pass this to the interpreter. However it is not possible to determine this name in all cases.‡

1.2. General features of object code

Pascal object code is relocatable as all addressing references for control transfers within the code are relative. The code consists of instructions interspersed with inline data. All instructions have a length that is an even number of bytes. No variables are kept in the object code area.

The first byte of a Pascal interpreter instruction contains an operation code. This allows a total of 256 major operation codes, and 232 of these are in use in the current *px*. The second byte of each interpreter instruction is called the "sub-operation code", or more commonly the *sub-opcode*. It contains a small integer that may, for example, be used as a block-structure level for the associated operation. If the instruction can take a longword constant, this constant is often packed into the sub-opcode if it fits into 8 bits and is not zero. A sub-opcode value of zero specifies that the constant would not fit and therefore follows in the next word. This is a space optimization, the value of zero for flagging the longer case being convenient because it is easy to test.

Other instruction formats are used. The branching instructions take an offset in the following word, operators that load constants onto the stack take arbitrarily long inline constant values, and many operations deal exclusively with data on the interpreter stack, requiring no inline data.

---

† UNIX is a trademark of Bell Laboratories.

‡ For instance, if the *pxref* program is placed in the directory '/usr/bin' then when the user types "pxref program.p" the first argument to the program, nominally the programs name, is "pxref." While it would be possible to search in the standard place, i.e. the current directory, and the system directories '/bin' and '/usr/bin' for a corresponding object file, this would be expensive and not guaranteed to succeed. Several shells exist that allow other directories to be searched for commands, and there is, in general, no way to determine what these directories are.

1.3. Stack structure of the interpreter

The interpreter emulates a stack-structured Pascal machine. The "load" instructions put values onto the stack, where all arithmetic operations take place. The "store" instructions take values off the stack and place them in an address that is also contained on the stack. The only way to move data or to compute in the machine is with the stack.

To make the interpreter operations more powerful and to thereby increase the interpreter speed, the arithmetic operations in the interpreter are "typed". That is, length conversion of arithmetic values occurs when they are used in an operation. This eliminates interpreter cycles for length conversion and the associated overhead. For example, when adding an integer that fits in one byte to one that requires four bytes to store, no "conversion" operators are required. The one byte integer is loaded onto the stack, followed by the four byte integer, and then an adding operator is used that has, implicit in its definition, the sizes of the arguments.

1.4. Data types in the interpreter

The interpreter deals with several different fundamental data types. In the memory of the machine, 1, 2, and 4 byte integers are supported, with only 2 and 4 byte integers being present on the stack. The interpreter always converts to 4 byte integers when there is a possibility of overflowing the shorter formats. This corresponds to the Pascal language definition of overflow in arithmetic operations that requires that the result be correct if all partial values lie within the bounds of the base integer type: 4 byte integer values.

Character constants are treated similarly to 1 byte integers for most purposes, as are Boolean values. All enumerated types are treated as integer values of an appropriate length, usually 1 byte. The interpreter also has real numbers, occupying 8 bytes of storage, and sets and strings of varying length. The appropriate operations are included for each data type, such as set union and intersection and an operation to write a string.

No special **packed** data formats are supported by the interpreter. The smallest unit of storage occupied by any variable is one byte. The built-ins *pack* and *unpack* thus degenerate to simple memory to memory transfers with no special processing.

1.5. Runtime environment

The interpreter runtime environment uses a stack data area and a heap data area, that are kept at opposite ends of memory and grow towards each other. All global variables and variables local to procedures and functions are kept in the stack area. Dynamically allocated variables and buffers for input/output are allocated in the heap.

The addressing of block structured variables is done by using a fixed display that contains the address of its stack frame for each statically active block.† This display is referenced by instructions that load and store variables and maintained by the operations for block entry and exit, and for non-local **goto** statements.

1.6. Dp, lc, loop

Three "global" variables in the interpreter, in addition to the "display", are the *dp*, *lc*, and the *loop*. The *dp* is a pointer to the display entry for the current block; the *lc* is the abstract machine location counter; and the *loop* is a register that holds the address of the main interpreter loop so that returning to the loop to fetch the next instruction is a fast operation.

1.7. The stack frame structure

Each active block has a stack frame consisting of three parts: a block mark, local variables, and temporary storage for partially evaluated expressions. The stack in the interpreter grows from the high addresses in memory to the low addresses, so that those parts of the stack frame that are

† Here "block" is being used to mean any *procedure*, *function* or the main program.

"on the top" of the stack have the most negative offsets from the display entry for the block. The major parts of the stack frame are represented in Figure 1.1.

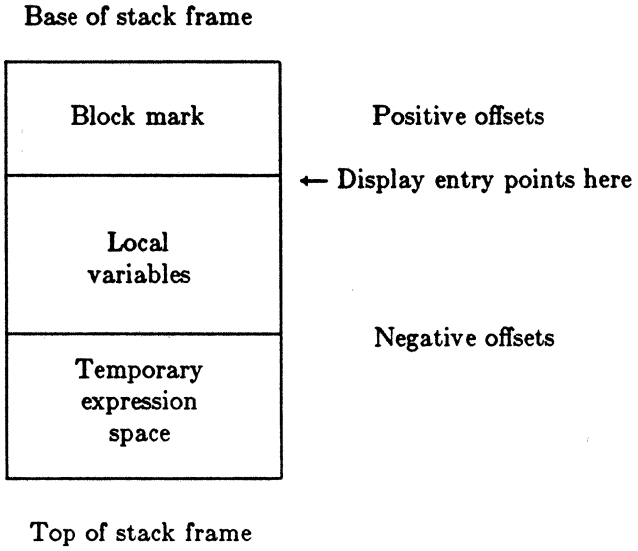


Figure 1.1 – Structure of stack frame

Note that the local variables of each block have negative offsets from the corresponding display entry, the "first" local variable having offset '–2'.

1.8. The block mark

The block mark contains the saved information necessary to restore the environment when the current block exits. It consists of two parts. The first and top-most part is saved by the CALL instruction in the interpreter. This information is not present for the main program as it is never "called". The second part of the block mark is created by the BEG begin block operator that also allocates and clears the local variable storage. The format of these blocks is represented in Figure 1.2.

The data saved by the CALL operator includes the line number *lino* of the point of call, that is printed if the program execution ends abnormally; the location counter *lc* giving the return address; and the current display entry address *dp* at the time of call.

The BEG begin operator saves the previous display contents at the level of this block, so that the display can be restored on block exit. A pointer to the beginning line number and the name of this block is also saved. This information is stored in the interpreter object code in-line after the BEG operator. It is used in printing a post-mortem backtrace. The saved file name and buffer reference are necessary because of the input/output structure (this is discussed in detail in sections 3.3 and 3.4). The top of stack reference gives the value the stack pointer should have when there are no expression temporaries on the stack. It is used for a consistency check in the LINO line number operators in the interpreter, that occurs before each statement executed. This helps to catch bugs in the interpreter, that often manifest themselves by leaving the stack non-empty between statements.

Note that there is no explicit static link here. Thus to set up the display correctly after a non-local **goto** statement one must "unwind" through all the block marks on the stack to rebuild the display.

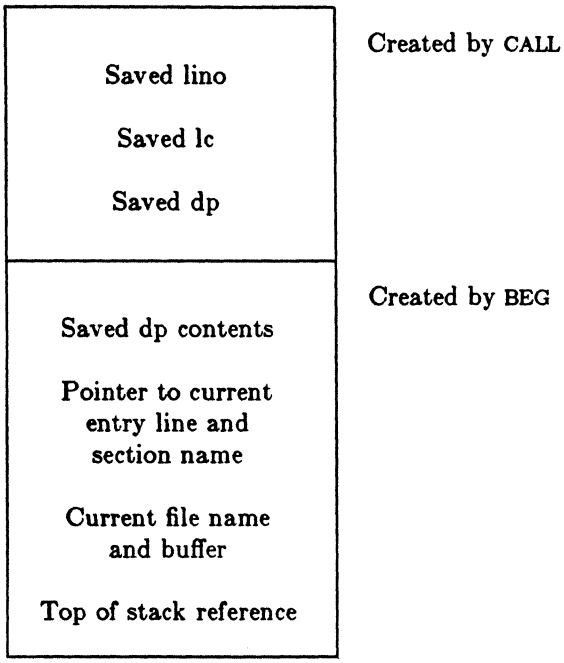


Figure 1.2 – Block mark structure

1.9. Arguments and return values

A function returns its value into a space reserved by the calling block. Arguments to a **function** are placed on top of this return area. For both **procedure** and **function** calls, arguments are placed at the end of the expression evaluation area of the caller. When a **function** completes, expression evaluation can continue after popping the arguments to the **function** off the stack, exactly as if the function value had been "loaded". The arguments to a **procedure** are also popped off the stack by the caller after its execution ends.

As a simple example consider the following stack structure for a call to a function *f*, of the form "f(a)".

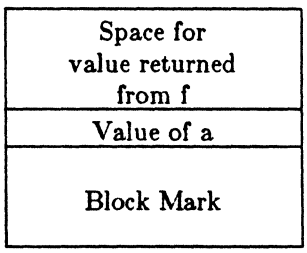


Figure 1.3 – Stack structure on function call 'f(a)'

If we suppose that *f* returns a *real* and that *a* is an integer, the calling sequence for this function would be:

```
PUSH      -8
RV4:I     a
CALL:I    f
POP       4
```

Here we use the operator PUSH to clear space for the return value, load *a* on the stack with a "right value" operator, call the function, pop off the argument *a*, and can then complete evaluation of the containing expression. The operations used here will be explained in section 2.

If the function *f* were given by

```
10 function f(i: integer): real;
11 begin
12     f := i
13 end;
```

then *f* would have code sequence:

```
BEG:2       0
            11
            "f"
LV:l        40
RV4:l       32
AS48
END
```

Here the BEG operator takes 9 bytes of inline data. The first byte specifies the length of the function name. The second longword specifies the amount of local variable storage, here none. The succeeding two lines give the line number of the **begin** and the name of the block for error traceback. The BEG operator places a name pointer in the block mark. The body of the **function** first takes an address of the **function** result variable *f* using the address of operator LV *a*. The next operation in the interpretation of this function is the loading of the value of *i*. *I* is at the level of the **function** *f*, here symbolically *l*, and the first variable in the local variable area. The **function** completes by assigning the 4 byte integer on the stack to the 8 byte return location, hence the AS48 assignment operator, and then uses the END operator to exit the current block.

1.10. The main interpreter loop

The main interpreter loop is simply:

```
iloop:
        caseb   (lc)+,$0,$255
        <table of opcode interpreter addresses>
```

The main opcode is extracted from the first byte of the instruction and used to index into the table of opcode interpreter addresses. Control is then transferred to the specified location. The sub-opcode may be used to index the display, as a small constant, or to specify one of several relational operators. In the cases where a constant is needed, but it is not small enough to fit in the byte sub-operator, a zero is placed there and the constant follows in the next word. Zero is easily tested for, as the instruction that fetches the sub-opcode sets the condition code flags. A construction like:

```
_OPER:
        cvtbl   (lc)+,r0
        bneq    L1
        cvtwl   (lc)+,r0
L1:     ...
```

is all that is needed to effect this packing of data. This technique saves space in the Pascal *obj* object code.

The address of the instruction at *iloop* is always contained in the register variable *loop*. Thus a return to the main interpreter is simply:

jmp (loop)

that is both quick and occupies little space.

1.11. Errors

Errors during interpretation fall into three classes:

1) Interpreter detected errors.
2) Hardware detected errors.
3) External events.

Interpreter detected errors include I/O errors and built-in function errors. These errors cause a subroutine call to an error routine with a single parameter indicating the cause of the error. Hardware errors such as range errors and overflows are fielded by a special routine that determines the opcode that caused the error. It then calls the error routine with an appropriate error parameter. External events include interrupts and system limits such as available memory. They generate a call to the error routine with an appropriate error code. The error routine processes the error condition, printing an appropriate error message and usually a backtrace from the point of the error.

2. Operations

2.1. Naming conventions and operation summary

Table 2.1 outlines the opcode typing convention. The expression "a above b" means that 'a' is on top of the stack with 'b' below it. Table 2.3 describes each of the opcodes. The character '*' at the end of a name specifies that all operations with the root prefix before the '*' are summarized by one entry. Table 2.2 gives the codes used to describe the type inline data expected by each instruction.

Table 2.1 – Operator Suffixes

Unary operator suffixes

Suffix	Example	Argument type
2	NEG2	Short integer (2 bytes)
4	SQR4	Long integer (4 bytes)
8	ABS8	Real (8 bytes)

Binary operator suffixes

Suffix	Example	Argument type
2	ADD2	Two short integers
24	MUL24	Short above long integer
42	REL42	Long above short integer
4	DIV4	Two long integers
28	DVD28	Short integer above real
48	REL48	Long integer above real
82	SUB82	Real above short integer
84	MUL84	Real above long integer
8	ADD8	Two reals

Other Suffixes

Suffix	Example	Argument types
T	ADDT	Sets
G	RELG	Strings

Table 2.2 – Inline data type codes

Code	Description
a	An address offset is given in the word following the instruction.
A	An address offset is given in the four bytes following the instruction.
l	An index into the display is given in the sub-opcode.
r	A relational operator is encoded in the sub-opcode. (see section 2.3)
s	A small integer is placed in the sub-opcode, or in the next word if it is zero or too large.
v	Variable length inline data.
w	A word value in the following word.
W	A long value in the following four bytes.
"	An inline constant string.

Table 2.3 – Machine operations

Mnemonic	Reference	Description
ABS*	2.7	Absolute value
ADD*	2.7	Addition
AND	2.4	Boolean and
ARGC	2.14	Returns number of arguments to current process
ARGV	2.14	Copy specified process argument into char array
AS*	2.5	Assignment operators
ASRT	2.12	Assert *true* to continue
ATAN	2.13	Returns arctangent of argument
BEG s,W,w,"	2.2,1.8	Write second part of block mark, enter block
BUFF	3.11	Specify buffering for file "output"
CALL l,A	2.2,1.8	Procedure or function call
CARD s	2.11	Cardinality of set
CASEOP*	2.9	Case statements
CHR*	2.15	Returns integer to ascii mapping of argument
CLCK	2.14	Returns user time of program
CON* v	2.5	Load constant operators
COS	2.13	Returns cos of argument
COUNT w	2.10	Count a statement count point
CTTOT s,w,w	2.11	Construct set
DATE	2.14	Copy date into char array
DEFNAME	3.11	Attach file name for **program** statement files
DISPOSE	2.15	Dispose of a heap allocation
DIV*	2.7	Fixed division
DVD*	2.7	Floating division
END	2.2,1.8	End block execution
EOF	3.10	Returns *true* if end of file
EOLN	3.10	Returns *true* if end of line on input text file
EXP	2.13	Returns exponential of argument
EXPO	2.13	Returns machine representation of real exponent
FILE	3.9	Push descriptor for active file
FLUSH	3.11	Flush a file
FNIL	3.7	Check file initialized, not eof, synced
FOR* a	2.12	For statements
GET	3.7	Get next record from a file
GOTO l,A	2.2,1.8	Non-local goto statement
HALT	2.2	Produce control flow backtrace
IF a	2.3	Conditional transfer
IN s,w,w	2.11	Set membership
INCT	2.11	Membership in a constructed set
IND*	2.6	Indirection operators
INX* s,w,w	2.6	Subscripting (indexing) operator
ITOD	2.12	Convert integer to real
ITOS	2.12	Convert integer to short integer
LINO s	2.2	Set line number, count statements
LLIMIT	2.14	Set linelimit for output text file
LLV l,W	2.6	Address of operator
LN	2.13	Returns natural log of argument
LRV* l,A	2.5	Right value (load) operators
LV l,w	2.6	Address of operator
MAX s,w	3.8	Maximum of top of stack and w
MESSAGE	3.6	Write to terminal
MIN s	3.8	Minimum of top of stack and s

Table 2.3 – Machine operations

Mnemonic	Reference	Description
MOD*	2.7	Modulus
MUL*	2.7	Multiplication
NAM A	3.8	Convert enumerated type value to print format
NEG*	2.7	Negation
NEW s	2.15	Allocate a record on heap, set pointer to it
NIL	2.6	Assert non-nil pointer
NODUMP s,W,w,"	2.2	BEG main program, suppress dump
NOT	2.4	Boolean not
ODD*	2.15	Returns *true* if argument is odd, *false* if even
OFF s	2.5	Offset address, typically used for field reference
OR	2.4	Boolean or
PACK s,w,w,w	2.15	Convert and copy from unpacked to packed
PAGE	3.8	Output a formfeed to a text file
POP s	2.2,1.9	Pop (arguments) off stack
PRED*	2.7	Returns predecessor of argument
PUSH s	2.2,1.9	Clear space (for function result)
PUT	3.8	Output a record to a file
PXPBUF w	2.10	Initialize *pxp* count buffer
RANDOM	2.13	Returns random number
RANG* v	2.8	Subrange checking
READ*	3.7	Read a record from a file
REL* r	2.3	Relational test yielding Boolean result
REMOVE	3.11	Remove a file
RESET	3.11	Open file for input
REWRITE	3.11	Open file for output
ROUND	2.13	Returns TRUNC(argument + 0.5)
RV* l,a	2.5	Right value (load) operators
SCLCK	2.14	Returns system time of program
SDUP	2.2	Duplicate top stack word
SEED	2.13	Set random seed, return old seed
SIN	2.13	Returns sin of argument
SQR*	2.7	Squaring
SQRT	2.13	Returns square root of argument
STLIM	2.14	Set program statement limit
STOD	2.12	Convert short integer to real
STOI	2.12	Convert short to long integer
SUB*	2.7	Subtraction
SUCC*	2.7	Returns successor of argument
TIME	2.14	Copy time into char array
TRA a	2.2	Short control transfer (local branching)
TRA4 A	2.2	Long control transfer
TRACNT w,A	2.10	Count a procedure entry
TRUNC	2.13	Returns integer part of argument
UNDEF	2.15	Returns *false*
UNIT*	3.10	Set active file
UNPACK s,w,w,w	2.15	Convert and copy from packed to unpacked
WCLCK	2.14	Returns current time stamp
WRITEC	3.8	Character unformatted write
WRITEF l	3.8	General formatted write
WRITES l	3.8	String unformatted write
WRITLN	3.8	Output a newline to a text file

2.2. Basic control operations

HALT

Corresponds to the Pascal procedure *halt*; causes execution to end with a post-mortem backtrace as if a run-time error had occurred.

BEG s,W,w,"

Causes the second part of the block mark to be created, and W bytes of local variable space to be allocated and cleared to zero. Stack overflow is detected here. w is the first line of the body of this section for error traceback, and the inline string (length s) the character representation of its name.

NODUMP s,W,w,"

Equivalent to BEG, and used to begin the main program when the "p" option is disabled so that the post-mortem backtrace will be inhibited.

END

Complementary to the operators CALL and BEG, exits the current block, calling the procedure *pclose* to flush buffers for and release any local files. Restores the environment of the caller from the block mark. If this is the end for the main program, all files are *flushed*, and the interpreter is exited.

CALL l,A

Saves the current line number, return address, and active display entry pointer *dp* in the first part of the block mark, then transfers to the entry point given by the relative address A, that is the beginning of a **procedure** or **function** at level l.

PUSH s

Clears s bytes on the stack. Used to make space for the return value of a **function** just before calling it.

POP s

Pop s bytes off the stack. Used after a **function** or **procedure** returns to remove the arguments from the stack.

TRA a

Transfer control to relative address a as a local **goto** or part of a structured statement.

TRA4 A

Transfer control to an absolute address as part of a non-local **goto** or to branch over procedure bodies.

LINO s

Set current line number to s. For consistency, check that the expression stack is empty as it should be (as this is the start of a statement.) This consistency check will fail only if there is a bug in the interpreter or the interpreter code has somehow been damaged. Increment the statement count and if it exceeds the statement limit, generate a fault.

GOTO l,A

Transfer control to address *A* that is in the block at level *l* of the display. This is a non-local **goto.** Causes each block to be exited as if with END, flushing and freeing files with *pclose,* until the current display entry is at level *l.*

SDUP*

Duplicate the word or long on the top of the stack. This is used mostly for constructing sets. See section 2.11.

2.3. If and relational operators

IF a

The interpreter conditional transfers all take place using this operator that examines the Boolean value on the top of the stack. If the value is *true*, the next code is executed, otherwise control transfers to the specified address.

REL* r

These take two arguments on the stack, and the sub-operation code specifies the relational operation to be done, coded as follows with 'a' above 'b' on the stack:

Code	Operation
0	a = b
2	a <> b
4	a < b
6	a > b
8	a <= b
10	a >= b

Each operation does a test to set the condition code appropriately and then does an indexed branch based on the sub-operation code to a test of the condition here specified, pushing a Boolean value on the stack.

Consider the statement fragment:

if a = b **then**

If *a* and *b* are integers this generates the following code:

```
RV4:l       a
RV4:l       b
REL4        =
IF          Else part offset

    ... Then part code ...
```

2.4. Boolean operators

The Boolean operators AND, OR, and NOT manipulate values on the top of the stack. All Boolean values are kept in single bytes in memory, or in single words on the stack. Zero represents a Boolean *false*, and one a Boolean *true*.

2.5. Right value, constant, and assignment operators

LRV* l,A
RV* l,a

The right value operators load values on the stack. They take a block number as a sub-opcode and load the appropriate number of bytes from that block at the offset specified in the following word onto the stack. As an example, consider LRV4:

```
_LRV4:
        cvtbl   (lc)+,r0                #r0 has display index
        addl3   _display(r0),(lc)+,r1   #r1 has variable address
        pushl   (r1)                    #put value on the stack
        jmp     (loop)
```

Here the interpreter places the display level in r0. It then adds the appropriate display value to the inline offset and pushes the value at this location onto the stack. Control then returns to the main interpreter loop. The RV* operators have short inline data that reduces the space required to address the first 32K of stack space in each stack frame. The operators RV14 and RV24 provide explicit conversion to long as the data is pushed. This saves the generation of STOI to align arguments to C subroutines.

CON* r

The constant operators load a value onto the stack from inline code. Small integer values are condensed and loaded by the CON1 operator, that is given by

```
_CON1:
        cvtbw   (lc)+,-(sp)
        jmp     (loop)
```

Here note that little work was required as the required constant was available at (lc)+. For longer constants, *lc* must be incremented before moving the constant. The operator CON takes a length specification in the sub-opcode and can be used to load strings and other variable length data onto the stack. The operators CON14 and CON24 provide explicit conversion to long as the constant is pushed.

AS*

The assignment operators are similar to arithmetic and relational operators in that they take two operands, both in the stack, but the lengths given for them specify first the length of the value on the stack and then the length of the target in memory. The target address in memory is under the value to be stored. Thus the statement

```
i := 1
```

where *i* is a full-length, 4 byte, integer, will generate the code sequence

```
LV:l        i
CON1:1
AS24
```

Here LV will load the address of *i*, that is really given as a block number in the sub-opcode and an offset in the following word, onto the stack, occupying a single word. CON1, that is a single word instruction, then loads the constant 1, that is in its sub-opcode, onto the stack. Since there are not one byte constants on the stack, this becomes a 2 byte, single word integer. The interpreter then assigns a length 2 integer to a length 4 integer using AS24. The code sequence for AS24 is given by:

```
_AS24:
        incl    lc
        cvtwl   (sp)+,*(sp)+
        jmp     (loop)
```

Thus the interpreter gets the single word off the stack, extends it to be a 4 byte integer gets the target address off the stack, and finally stores the value in the target. This is a typical use of the constant and assignment operators.

2.6. Addressing operations

LLV l,W
LV l,w

The most common operation done by the interpreter is the "left value" or "address of" operation. It is given by:

```
_LLV:
        cvtbl   (lc)+,r0                #r0 has display index
        addl3   _display(r0),(lc)+,-(sp) #push address onto the stack
        jmp     (loop)
```

It calculates an address in the block specified in the sub-opcode by adding the associated display entry to the offset that appears in the following word. The LV operator has a short inline data that reduces the space required to address the first 32K of stack space in each call frame.

OFF s

The offset operator is used in field names. Thus to get the address of

```
p↑.f1
```

pi would generate the sequence

```
RV:l    p
OFF     f1
```

where the RV loads the value of *p*, given its block in the sub-opcode and offset in the following word, and the interpreter then adds the offset of the field *f1* in its record to get the correct address. OFF takes its argument in the sub-opcode if it is small enough.

NIL

The example above is incomplete, lacking a check for a **nil** pointer. The code generated would be

```
RV:l    p
NIL
OFF     f1
```

where the NIL operation checks for a *nil* pointer and generates the appropriate runtime error if it is.

LVCON s,"

A pointer to the specified length inline data is pushed onto the stack. This is primarily used for *printf* type strings used by WRITEF. (see sections 3.6 and 3.8)

INX* s,w,w

The operators INX2 and INX4 are used for subscripting. For example, the statement

a[i] := 2.0

with *i* an integer and *a* an "array [1..1000] of real" would generate

```
LV:l      a
RV4:l     i
INX4:8    1,999
CON8      2.0
AS8
```

Here the LV operation takes the address of *a* and places it on the stack. The value of *i* is then placed on top of this on the stack. The array address is indexed by the length 4 index (a length 2 index would use INX2) where the individual elements have a size of 8 bytes. The code for INX4 is:

```
_INX4:
        cvtbl   (lc)+,r0
        bneq    L1
        cvtwl   (lc)+,r0            #r0 has size of records
L1:
        cvtwl   (lc)+,r1            #r1 has lower bound
        movzwl  (lc)+,r2            #r2 has upper-lower bound
        subl3   r1,(sp)+,r3         #r3 has base subscript
        cmpl    r3,r2               #check for out of bounds
        bgtru   esubscr
        mull2   r0,r3               #calculate byte offset
        addl2   r3,(sp)             #calculate actual address
        jmp     (loop)
esubscr:
        movw    $ESUBSCR,_perrno
        jbr     error
```

Here the lower bound is subtracted, and range checked against the upper minus lower bound. The offset is then scaled to a byte offset into the array and added to the base address on the stack. Multi-dimension subscripts are translated as a sequence of single subscriptings.

IND*

For indirect references through **var** parameters and pointers, the interpreter has a set of indirection operators that convert a pointer on the stack into a value on the stack from that address. different IND operators are necessary because of the possibility of different length operands. The IND14 and IND24 operators do conversions to long as they push their data.

2.7. Arithmetic operators

The interpreter has many arithmetic operators. All operators produce results long enough to prevent overflow unless the bounds of the base type are exceeded. The basic operators available are

Addition: ADD*, SUCC*
Subtraction: SUB*, PRED*
Multiplication: MUL*, SQR*
Division: DIV*, DVD*, MOD*
Unary: NEG*, ABS*

2.8. Range checking

The interpreter has several range checking operators. The important distinction among these operators is between values whose legal range begins at zero and those that do not begin at zero, for example a subrange variable whose values range from 45 to 70. For those that begin at zero, a simpler "logical" comparison against the upper bound suffices. For others, both the low and upper bounds must be checked independently, requiring two comparisons. On the VAX 11/780 both checks are done using a single index instruction so the only gain is in reducing the inline data.

2.9. Case operators

The interpreter includes three operators for **case** statements that are used depending on the width of the **case** label type. For each width, the structure of the case data is the same, and is represented in figure 2.4.

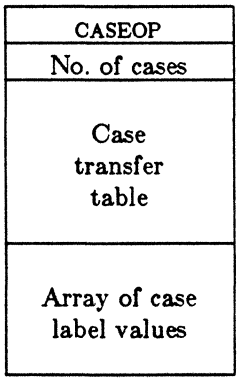


Figure 2.4 – Case data structure

The CASEOP case statement operators do a sequential search through the case label values. If they find the label value, they take the corresponding entry from the transfer table and cause the interpreter to branch to the specified statement. If the specified label is not found, an error results.

The CASE operators take the number of cases as a sub-opcode if possible. Three different operators are needed to handle single byte, word, and long case transfer table values. For example, the CASEOP1 operator has the following code sequence:

```
_CASEOP1:
        cvtbl   (lc)+,r0
        bneq    L1
        cvtwl   (lc)+,r0                #r0 has length of case table
L1:
        movaw   (lc)[r0],r2             #r2 has pointer to case labels
        movzwl  (sp)+,r3                #r3 has the element to find
        locc    r3,r0,(r2)              #r0 has index of located element
        beql    caserr                  #element not found
        mnegl   r0,r0                   #calculate new lc
        cvtwl   (r2)[r0],r1             #r1 has lc offset
        addl2   r1,lc
        jmp     (loop)
caserr:
        movw    $ECASE,_perrno
        jbr     error
```

Here the interpreter first computes the address of the beginning of the case label value area by adding twice the number of case label values to the address of the transfer table, since the transfer table entries are 2 byte address offsets. It then searches through the label values, and generates an ECASE error if the label is not found. If the label is found, the index of the corresponding entry in the transfer table is extracted and that offset is added to the interpreter location counter.

2.10. Operations supporting pxp

The following operations are defined to do execution profiling.

PXPBUF w

Causes the interpreter to allocate a count buffer with w four byte counters and to clear them to zero. The count buffer is placed within an image of the *pmon.out* file as described in the *PXP Implementation Notes*. The contents of this buffer are written to the file *pmon.out* when the program ends.

COUNT w

Increments the counter specified by w.

TRACNT w,A

Used at the entry point to procedures and functions, combining a transfer to the entry point of the block with an incrementing of its entry count.

2.11. Set operations

The set operations: union ADDT, intersection MULT, element removal SUBT, and the set relationals RELT are straightforward. The following operations are more interesting.

CARD s

Takes the cardinality of a set of size s bytes on top of the stack, leaving a 2 byte integer count. CARD uses the **ffs** opcode to successively count the number of set bits in the set.

CTTOT s,w,w

Constructs a set. This operation requires a non-trivial amount of work, checking bounds and setting individual bits or ranges of bits. This operation sequence is slow, and motivates the presence of the operator INCT below. The arguments to CTTOT include the number of

elements *s* in the constructed set, the lower and upper bounds of the set, the two *w* values, and a pair of values on the stack for each range in the set, single elements in constructed sets being duplicated with SDUP to form degenerate ranges.

IN s,w,w

The operator **in** for sets. The value *s* specifies the size of the set, the two *w* values the lower and upper bounds of the set. The value on the stack is checked to be in the set on the stack, and a Boolean value of *true* or *false* replaces the operands.

INCT

The operator **in** on a constructed set without constructing it. The left operand of **in** is on top of the stack followed by the number of pairs in the constructed set, and then the pairs themselves, all as single word integers. Pairs designate runs of values and single values are represented by a degenerate pair with both value equal. This operator is generated in grammatical constructs such as

if character **in** [`+´, `-´, `*´, `/´]

or

if character **in** [`a´..`z´, `$´, `_´]

These constructs are common in Pascal, and INCT makes them run much faster in the interpreter, as if they were written as an efficient series of **if** statements.

2.12. Miscellaneous

Other miscellaneous operators that are present in the interpreter are ASRT that causes the program to end if the Boolean value on the stack is not *true*, and STOI, STOD, ITOD, and ITOS that convert between different length arithmetic operands for use in aligning the arguments in **procedure** and **function** calls, and with some untyped built-ins, such as SIN and COS.

Finally, if the program is run with the run-time testing disabled, there are special operators for **for** statements and special indexing operators for arrays that have individual element size that is a power of 2. The code can run significantly faster using these operators.

2.13. Mathematical Functions

The transcendental functions SIN, COS, ATAN, EXP, LN, SQRT, SEED, and RANDOM are taken from the standard UNIX mathematical package. These functions take double precision floating point values and return the same.

The functions EXPO, TRUNC, and ROUND take a double precision floating point number. EXPO returns an integer representing the machine representation of its argument's exponent, TRUNC returns the integer part of its argument, and ROUND returns the rounded integer part of its argument.

2.14. System functions and procedures

LLIMIT

A line limit and a file pointer are passed on the stack. If the limit is non-negative the line limit is set to the specified value, otherwise it is set to unlimited. The default is unlimited.

STLIM

A statement limit is passed on the stack. The statement limit is set as specified. The default is 500,000. No limit is enforced when the "p" option is disabled.

CLCK
SCLCK

CLCK returns the number of milliseconds of user time used by the program; SCLCK returns the number of milliseconds of system time used by the program.

WCLCK

The number of seconds since some predefined time is returned. Its primary usefulness is in determining elapsed time and in providing a unique time stamp.

The other system time procedures are DATE and TIME that copy an appropriate text string into a pascal string array. The function ARGC returns the number of command line arguments passed to the program. The procedure ARGV takes an index on the stack and copies the specified command line argument into a pascal string array.

2.15. Pascal procedures and functions

PACK s,w,w,w
UNPACK s,w,w,w

They function as a memory to memory move with several semantic checks. They do no "unpacking" or "packing" in the true sense as the interpreter supports no packed data types.

NEW s
DISPOSE s

An LV of a pointer is passed. NEW allocates a record of a specified size and puts a pointer to it into the pointer variable. DISPOSE deallocates the record pointed to by the pointer and sets the pointer to NIL.

The function CHR* converts a suitably small integer into an ascii character. Its primary purpose is to do a range check. The function ODD* returns *true* if its argument is odd and returns *false* if its argument is even. The function UNDEF always returns the value *false*.

3. Input/output

3.1. The files structure

Each file in the Pascal environment is represented by a pointer to a *files* structure in the heap. At the location addressed by the pointer is the element in the file's window variable. Behind this window variable is information about the file, at the following offsets:

-108	FNAME	Text name of associated UNIX file
-30	LCOUNT	Current count of lines output
-26	LLIMIT	Maximum number of lines permitted
-22	FBUF	UNIX FILE pointer
-18	FCHAIN	Chain to next file
-14	FLEV	Pointer to associated file variable
-10	PFNAME	Pointer to name of file for error messages
-6	FUNIT	File status flags
-4	FSIZE	Size of elements in the file
0		File window element

Here FBUF is a pointer to the system FILE block for the file. The standard system I/O library is used that provides block buffered input/output, with 1024 characters normally transferred at each read or write.

The files in the Pascal environment, are all linked together on a single file chain through the FCHAIN links. For each file the FLEV pointer gives its associated file variable. These are used to free files at block exit as described in section 3.3 below.

The FNAME and PFNAME give the associated file name for the file and the name to be used when printing error diagnostics respectively. Although these names are usually the same, *input* and *output* usually have no associated file name so the distinction is necessary.

The FUNIT word contains a set of flags. whose representations are:

EOF	0x0100	At end-of-file
EOLN	0x0200	At end-of-line (text files only)
SYNC	0x0400	File window is out of sync
TEMP	0x0800	File is temporary
FREAD	0x1000	File is open for reading
FWRITE	0x2000	File is open for writing
FTEXT	0x4000	File is a text file; process EOLN
FDEF	0x8000	File structure created, but file not opened

The EOF and EOLN bits here reflect the associated built-in function values. TEMP specifies that the file has a generated temporary name and that it should therefore be removed when its block exits. FREAD and FWRITE specify that *reset* and *rewrite* respectively have been done on the file so that input or output operations can be done. FTEXT specifies the file is a text file so that EOLN processing should be done, with newline characters turned into blanks, etc.

The SYNC bit, when true, specifies that there is no usable image in the file buffer window. As discussed in the *Berkeley Pascal User's Manual,* the interactive environment necessitates having "input^" undefined at the beginning of execution so that a program may print a prompt before the user is required to type input. The SYNC bit implements this. When it is set, it specifies that the element in the window must be updated before it can be used. This is never done until necessary.

3.2. Initialization of files

All the variables in the Pascal runtime environment are cleared to zero on block entry. This is necessary for simple processing of files. If a file is unused, its pointer will be **nil.** All references to an inactive file are thus references through a **nil** pointer. If the Pascal system did not clear storage to zero before execution it would not be possible to detect inactive files in this simple way; it would probably be necessary to generate (possibly complicated) code to initialize each file on block entry.

When a file is first mentioned in a *reset* or *rewrite* call, a buffer of the form described above is associated with it, and the necessary information about the file is placed in this buffer. The file is also linked into the active file chain. This chain is kept sorted by block mark address, the FLEV entries.

3.3. Block exit

When block exit occurs the interpreter must free the files that are in use in the block and their associated buffers. This is simple and efficient because the files in the active file chain are sorted by increasing block mark address. This means that the files for the current block will be at the front of the chain. For each file that is no longer accessible the interpreter first flushes the files buffer if it is an output file. The interpreter then returns the file buffer and the files structure and window to the free space in the heap and removes the file from the active file chain.

3.4. Flushing

Flushing all the file buffers at abnormal termination, or on a call to the procedure *flush* or *message* is done by flushing each file on the file chain that has the FWRITE bit set in its flags word.

3.5. The active file

For input-output, *px* maintains a notion of an active file. Each operation that references a file makes the file it will be using the active file and then does its operation. A subtle point here is that one may do a procedure call to *write* that involves a call to a function that references another file, thereby destroying the active file set up before the *write*. Thus the active file is saved at block entry in the block mark and restored at block exit.†

3.6. File operations

Files in Pascal can be used in two distinct ways: as the object of *read, write, get,* and *put* calls, or indirectly as though they were pointers. The second use as pointers must be careful not to destroy the active file in a reference such as

```
write(output, input↑)
```

or the system would incorrectly write on the input device.

The fundamental operator related to the use of a file is FNIL. This takes the file variable, as a pointer, insures that the pointer is not **nil,** and also that a usable image is in the file window, by forcing the SYNC bit to be cleared.

A simple example that demonstrates the use of the file operators is given by

```
writeln(f)
```

that produces

```
RV:1        f
UNIT
WRITLN
```

3.7. Read operations

GET

Advance the active file to the next input element.

FNIL

A file pointer is on the stack. Insure that the associated file is active and that the file is synced so that there is input available in the window.

READ*

If the file is a text file, read a block of text and convert it to the internal type of the specified operand. If the file is not a text file then do an unformatted read of the next record. The procedure READLN reads upto and including the next end of line character.

READE A

The operator READE reads a string name of an enumerated type and converts it to its internal value. READE takes a pointer to a data structure as shown in figure 3.2.

† It would probably be better to dispense with the notion of active file and use another mechanism that did not involve extra overhead on each procedure and function call.

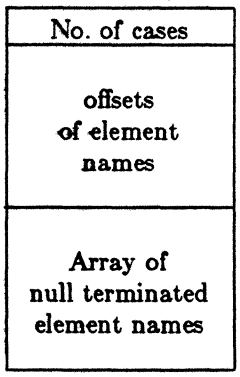


Figure 3.2 – Enumerated type conversion structure

See the description of NAM in the next section for an example.

3.8. Write operations

PUT

Output the element in the active file window.

WRITEF s

The argument(s) on the stack are output by the *fprintf* standard I/O library routine. The sub-opcode *s* specifies the number of longword arguments on the stack.

WRITEC

The character on the top of the stack is output without formatting. Formatted characters must be output with WRITEF.

WRITES

The string specified by the pointer on the top of the stack is output by the *fwrite* standard I/O library routine. All characters including nulls are printed.

WRITLN

A linefeed is output to the active file. The line-count for the file is incremented and checked against the line limit.

PAGE

A formfeed is output to the active file.

NAM A

The value on the top of the stack is converted to a pointer to an enumerated type string name. The address A points to an enumerated type structure identical to that used by READE. An error is raised if the value is out of range. The form of this structure for the predefined type **boolean** is shown in figure 3.3. The code for NAM is

bool:

2
6
12
17
"false"
"true"

Figure 3.3 – Boolean type conversion structure

```
_NAM:
        incl    lc
        addl3   (lc)+,ap,r6        #r6 points to scalar name list
        movl    (sp)+,r3            #r3 has data value
        cmpw    r3,(r6)+            #check value out of bounds
        bgequ   enamrng
        movzwl  (r6)[r3],r4         #r4 has string index
        pushab  (r6)[r4]            #push string pointer
        jmp     (loop)
enamrng:
        movw    $ENAMRNG,_perrno
        jbr     error
```

The address of the table is calculated by adding the base address of the interpreter code, *ap* to the offset pointed to by *lc*. The first word of the table gives the number of records and provides a range check of the data to be output. The pointer is then calculated as

```
tblbase = ap + A;
size = *tblbase++;
return(tblbase + tblbase[value]);
```

MAX s,w

The sub-opcode *s* is subtracted from the integer on the top of the stack. The maximum of the result and the second argument, *w*, replaces the value on the top of the stack. This function verifies that variable specified width arguments are non-negative, and meet certain minimum width requirements.

MIN s

The minimum of the value on the top of the stack and the sub-opcode replaces the value on the top of the stack.

The uses of files and the file operations are summarized in an example which outputs a real variable (r) with a variable width field (i).

```
writeln('r =',r:i,' ',true);
```

that generates the code

```
UNITOUT
FILE
CON14:1
CON14:3
LVCON:4     "r ="
WRITES
RV8:l       r
RV4:l       i
MAX:8       1
RV4:l       i
MAX:1       1
LVCON:8     " %*.*E"
FILE
WRITEF:6
CONC4       ' '
WRITEC
CON14:1
NAM         bool
LVCON:4     "%s"
FILE
WRITEF:3
WRITLN
```

Here the operator UNITOUT is an abbreviated form of the operator UNIT that is used when the file to be made active is *output*. A file descriptor, record count, string size, and a pointer to the constant string "r =" are pushed and then output by WRITES. Next the value of *r* is pushed on the stack and the precision size is calculated by taking seven less than the width, but not less than one. This is followed by the width that is reduced by one to leave space for the required leading blank. If the width is too narrow, it is expanded by *fprintf*. A pointer to the format string is pushed followed by a file descriptor and the operator WRITEF that prints out *r*. The value of six on WRITEF comes from two longs for *r* and a long each for the precision, width, format string pointer, and file descriptor. The operator CONC4 pushes the *blank* character onto a long on the stack that is then printed out by WRITEC. The internal representation for *true* is pushed as a long onto the stack and is then replaced by a pointer to the string "true" by the operator NAM using the table *bool* for conversion. This string is output by the operator WRITEF using the format string "%s". Finally the operator WRITLN appends a newline to the file.

3.9. File activation and status operations

UNIT*

The file pointed to by the file pointer on the top of the stack is converted to be the active file. The opcodes UNITINP and UNITOUT imply standard input and output respectively instead of explicitly pushing their file pointers.

FILE

The standard I/O library file descriptor associated with the active file is pushed onto the stack.

EOF

The file pointed to by the file pointer on the top of the stack is checked for end of file. A boolean is returned with *true* indicating the end of file condition.

EOLN

The file pointed to by the file pointer on the top of the stack is checked for end of line. A boolean is returned with *true* indicating the end of line condition. Note that only text files can check for end of line.

3.10. File housekeeping operations

DEFNAME

Four data items are passed on the stack; the size of the data type associated with the file, the maximum size of the file name, a pointer to the file name, and a pointer to the file variable. A file record is created with the specified window size and the file variable set to point to it. The file is marked as defined but not opened. This allows **program** statement association of file names with file variables before their use by a RESET or a REWRITE.

BUFF s

The sub-opcode is placed in the external variable *_bufopt* to specify the amount of I/O buffering that is desired. The current options are:

0 – character at a time buffering
1 – line at a time buffering
2 – block buffering

The default value is 1.

RESET
REWRITE

Four data items are passed on the stack; the size of the data type associated with the file, the maximum size of the name (possibly zero), a pointer to the file name (possibly null), and a pointer to the file variable. If the file has never existed it is created as in DEFNAME. If no file name is specified and no previous name exists (for example one created by DEFNAME) then a system temporary name is created. RESET then opens the file for input, while REWRITE opens the file for output.

The three remaining file operations are FLUSH that flushes the active file, REMOVE that takes the pointer to a file name and removes the specified file, and MESSAGE that flushes all the output files and sets the standard error file to be the active file.

4. Conclusions

It is appropriate to consider, given the amount of time invested in rewriting the interpreter, whether the time was well spent, or whether a code-generator could have been written with an equivalent amount of effort. The Berkeley Pascal system is being modified to interface to the code generator of the portable C compiler with not much more work than was involved in rewritting *px*. However this compiler will probably not supercede the interpreter in an instructional environment as the necessary loading and assembly processes will slow the compilation process to a noticeable degree. This effect will be further exaggerated because student users spend more time in compilation than in execution. Measurements over the course of a quarter at Berkeley with a mixture of students from beginning programming to upper division compiler construction show that the amount of time in compilation exceeds the amount of time spent in the interpreter, the ratio being approximately 60/40.

A more promising approach might have been a throw-away code generator such as was done for the WATFIV system. However the addition of high-quality post-mortem and interactive debugging facilities become much more difficult to provide than in the interpreter environment.

The Programming Language EFL

Stuart I. Feldman

Fortran
Preprocessors
Ratfor

ABSTRACT

EFL is a clean, general purpose computer language intended to encourage portable programming. It has a uniform and readable syntax and good data and control flow structuring. EFL programs can be translated into efficient Fortran code, so the EFL programmer can take advantage of the ubiquity of Fortran, the valuable libraries of software written in that language, and the portability that comes with the use of a standardized language, without suffering from Fortran's many failings as a language. It is especially useful for numeric programs. The EFL language permits the programmer to express complicated ideas in a comprehensible way, while permitting access to the power of the Fortran environment. EFL can be viewed as a descendant of B. W. Kernighan's Ratfor [1]; the name originally stood for 'Extended Fortran Language'. The current version of the EFL compiler is written in portable C.

4 June 1979

The Programming Language EFL

Stuart I. Feldman

Fortran
Preprocessors
Ratfor

1. INTRODUCTION

1.1. Purpose

EFL is a clean, general purpose computer language intended to encourage portable programming. It has a uniform and readable syntax and good data and control flow structuring. EFL programs can be translated into efficient Fortran code, so the EFL programmer can take advantage of the ubiquity of Fortran, the valuable libraries of software written in that language, and the portability that comes with the use of a standardized language, without suffering from Fortran's many failings as a language. It is especially useful for numeric programs. Thus, the EFL language permits the programmer to express complicated ideas in a comprehensible way, while permitting access to the power of the Fortran environment.

1.2. History

EFL can be viewed as a descendant of B. W. Kernighan's Ratfor [1]; the name originally stood for 'Extended Fortran Language'. A. D. Hall designed the initial version of the language and wrote a preliminary version of a compiler. I extended and modified the language and wrote a full compiler (in C) for it. The current compiler is much more than a simple preprocessor: it attempts to diagnose all syntax errors, to provide readable Fortran output, and to avoid a number of niggling restrictions. To achieve this goal, a sizable two-pass translator is needed.

1.3. Notation

In examples and syntax specifications, **boldface** type is used to indicate literal words and punctuation, such as **while**. Words in *italic* type indicate an item in a category, such as an *expression*. A construct surrounded by double brackets represents a list of one or more of those items, separated by commas. Thus, the notation

[[*item*]]

could refer to any of the following:

item
item, *item*
item, *item*, *item*

The reader should have a fair degree of familiarity with some procedural language. There will be occasional references to Ratfor and to Fortran which may be ignored if the reader is unfamiliar with those languages.

2. LEXICAL FORM

2.1. Character Set

The following characters are legal in an EFL program:

letters	**a b c d e f g h i j k l m**
	n o p q r s t u v w x y z
digits	**0 1 2 3 4 5 6 7 8 9**
white space	*blank tab*
quotes	' "
sharp	#
continuation	_
braces	{ }
parentheses	( )
other	, ; : . + - * /
	= < > & ~ | $

Letter case (upper or lower) is ignored except within strings, so '**a**' and '**A**' are treated as the same character. All of the examples below are printed in lower case. An exclamation mark ('!') may be used in place of a tilde ('~'). Square brackets ('[' and ']') may be used in place of braces ('{' and '}').

2.2. Lines

EFL is a line-oriented language. Except in special cases (discussed below), the end of a line marks the end of a token and the end of a statement. The trailing portion of a line may be used for a comment. There is a mechanism for diverting input from one source file to another, so a single line in the program may be replaced by a number of lines from the other file. Diagnostic messages are labeled with the line number of the file on which they are detected.

2.2.1. White Space

Outside of a character string or comment, any sequence of one or more spaces or tab characters acts as a single space. Such a space terminates a token.

2.2.2. Comments

A comment may appear at the end of any line. It is introduced by a sharp (#) character, and continues to the end of the line. (A sharp inside of a quoted string does not mark a comment.) The sharp and succeeding characters on the line are discarded. A blank line is also a comment. Comments have no effect on execution.

2.2.3. Include Files

It is possible to insert the contents of a file at a point in the source text, by referencing it in a line like

include joe

No statement or comment may follow an **include** on a line. In effect, the **include** line is replaced by the lines in the named file, but diagnostics refer to the line number in the included file. **Includes** may be nested at least ten deep.

2.2.4. Continuation

Lines may be continued explicitly by using the underscore (_) character. If the last character of a line (after comments and trailing white space have been stripped) is an underscore, the end of line and the initial blanks on the next line are ignored. Underscores are ignored in other contexts (except inside of quoted strings). Thus

```
1_000_000_
000
```

equals 10^9.

There are also rules for continuing lines automatically: the end of line is ignored whenever it is obvious that the statement is not complete. To be specific, a statement is continued if the last token on a line is an operator, comma, left brace, or left parenthesis. (A statement is not continued just because of unbalanced braces or parentheses.) Some compound statements are also continued automatically; these points are noted in the sections on executable statements.

2.2.5. Multiple Statements on a Line

A semicolon terminates the current statement. Thus, it is possible to write more than one statement on a line. A line consisting only of a semicolon, or a semicolon following a semicolon, forms a null statement.

2.3. Tokens

A program is made up of a sequence of tokens. Each token is a sequence of characters. A blank terminates any token other than a quoted string. End of line also terminates a token unless explicit continuation (see above) is signaled by an underscore.

2.3.1. Identifiers

An identifier is a letter or a letter followed by letters or digits. The following is a list of the reserved words that have special meaning in EFL. They will be discussed later.

array	**exit**	**precision**
automatic	**external**	**procedure**
break	**false**	**read**
call	**field**	**readbin**
case	**for**	**real**
character	**function**	**repeat**
common	**go**	**return**
complex	**goto**	**select**
continue	**if**	**short**
debug	**implicit**	**sizeof**
default	**include**	**static**
define	**initial**	**struct**
dimension	**integer**	**subroutine**
do	**internal**	**true**
double	**lengthof**	**until**
doubleprecision	**logical**	**value**
else	**long**	**while**
end	**next**	**write**
equivalence	**option**	**writebin**

The use of these words is discussed below. These words may not be used for any other purpose.

2.3.2. Strings

A character string is a sequence of characters surrounded by quotation marks. If the string is bounded by single-quote marks ('), it may contain double quote marks ("), and vice versa. A quoted string may not be broken across a line boundary.

```
'hello there'
"ain't misbehavin'"
```

2.3.3. Integer Constants

An integer constant is a sequence of one or more digits.

0
57
123456

2.3.4. Floating Point Constants

A floating point constant contains a dot and/or an exponent field. An *exponent field* is a letter **d** or **e** followed by an optionally signed integer constant. If *I* and *J* are integer constants and *E* is an exponent field, then a floating constant has one of the following forms:

.I
I.
I.J
IE
I.E
.IE
I.JE

2.3.5. Punctuation

Certain characters are used to group or separate objects in the language. These are

parentheses	( )
braces	{ }
comma	,
semicolon	;
colon	:
end-of-line	

The end-of-line is a token (statement separator) when the line is neither blank nor continued.

2.3.6. Operators

The EFL operators are written as sequences of one or more non-alphanumeric characters.

```
+  -  *  /  **
<  <=  >  >=  ==  ~=
&&  ||  &  |
+=  -=  /=  **=
&&=  ||=  &=  |=
->  .  $
```

A dot ('.') is an operator when it qualifies a structure element name, but not when it acts as a decimal point in a numeric constant. There is a special mode (see the Atavisms section) in which some of the operators may be represented by a string consisting of a dot, an identifier, and a dot (*e.g.*, **.lt.**).

2.4. Macros

EFL has a simple macro substitution facility. An identifier may be defined to be equal to a string of tokens; whenever that name appears as a token in the program, the string replaces it. A macro name is given a value in a **define** statement like

```
define count    n += 1
```

Any time the name **count** appears in the program, it is replaced by the statement

```
n += 1
```

A **define** statement must appear alone on a line; the form is

define *name rest-of-line*

Trailing comments are part of the string.

3. PROGRAM FORM

3.1. Files

A *file* is a sequence of lines. A file is compiled as a single unit. It may contain one or more procedures. Declarations and options that appear outside of a procedure affect the succeeding procedures on that file.

3.2. Procedures

Procedures are the largest grouping of statements in EFL. Each procedure has a name by which it is invoked. (The first procedure invoked during execution, known as the *main* procedure, has the null name.) Procedure calls and argument passing are discussed in Section 8.

3.3. Blocks

Statements may be formed into groups inside of a procedure. To describe the scope of names, it is convenient to introduce the ideas of *block* and of *nesting level*. The beginning of a program file is at nesting level zero. Any options, macro definitions, or variable declarations there are also at level zero. The text immediately following a **procedure** statement is at level 1. After the declarations, a left brace marks the beginning of a new block and increases the nesting level by 1; a right brace drops the level by 1. (Braces inside declarations do not mark blocks.) (See Section 7.2). An **end** statement marks the end of the procedure, level 1, and the return to level 0. A name (variable or macro) that is defined at level *k* is defined throughout that block and in all deeper nested levels in which that name is not redefined or redeclared. Thus, a procedure might look like the following:

```
#  block 0
procedure george
real x
x = 2
...
if(x > 2)
        {       # new block
        integer x  # a different variable
        do x = 1,7
                write(,x)
        ...
        }       # end of block
end     # end of procedure, return to block 0
```

3.4. Statements

A statement is terminated by end of line or by a semicolon. Statements are of the following types:

Option
Include
Define

Procedure
End

Declarative
Executable

The **option** statement is described in Section 10. The **include, define,** and **end** statements have been described above; they may not be followed by another statement on a line. Each procedure begins with a **procedure** statements and finishes with an **end** statement; these are discussed in Section 8. Declarations describe types and values of variables and procedures. Executable statements cause specific actions to be taken. A block is an example of an executable statement; it is made up of declarative and executable statements.

3.5. Labels

An executable statement may have a *label* which may be used in a branch statement. A label is an identifier followed by a colon, as in

```
          read(, x)
          if(x < 3) goto error
          ...
error:    fatal("bad input")
```

4. DATA TYPES AND VARIABLES

EFL supports a small number of basic (scalar) types. The programmer may define objects made up of variables of basic type; other aggregates may then be defined in terms of previously defined aggregates.

4.1. Basic Types

The basic types are

logical
integer
field(m**:**n**)**
real
complex
long real
long complex
character(n**)**

A logical quantity may take on the two values true and false. An integer may take on any whole number value in some machine-dependent range. A field quantity is an integer restricted to a particular closed interval ($[m:n]$). A 'real' quantity is a floating point approximation to a real or rational number. A long real is a more precise approximation to a rational. (Real quantities are represented as single precision floating point numbers; long reals are double precision floating point numbers.) A complex quantity is an approximation to a complex number, and is represented as a pair of reals. A character quantity is a fixed-length string of n characters.

4.2. Constants

There is a notation for a constant of each basic type.

A logical may take on the two values

true
false

An integer or field constant is a fixed point constant, optionally preceded by a plus or minus sign, as in

17
-94
+6
0

A long real ('double precision') constant is a floating point constant containing an exponent field that begins with the letter **d.** A real ('single precision') constant is any other floating point constant. A real or long real constant may be preceded by a plus or minus sign. The following are valid **real** constants:

17.3
-.4
7.9e-6 ($= 7.9 \times 10^{-6}$)
14e9 ($= 1.4 \times 10^{10}$)

The following are valid **long real** constants

7.9d-6 ($= 7.9 \times 10^{-6}$)
5d3

A character constant is a quoted string.

4.3. Variables

A variable is a quantity with a name and a location. At any particular time the variable may also have a value. (A variable is said to be *undefined* before it is initialized or assigned its first value, and after certain indefinite operations are performed.) Each variable has certain attributes:

4.3.1. Storage Class

The association of a name and a location is either transitory or permanent. Transitory association is achieved when arguments are passed to procedures. Other associations are permanent (static). (A future extension of EFL may include dynamically allocated variables.)

4.3.2. Scope of Names

The names of common areas are global, as are procedure names: these names may be used anywhere in the program. All other names are local to the block in which they are declared.

4.3.3. Precision

Floating point variables are either of normal or **long** precision. This attribute may be stated independently of the basic type.

4.4. Arrays

It is possible to declare rectangular arrays (of any dimension) of values of the same type. The index set is always a cross-product of intervals of integers. The lower and upper bounds of the intervals must be constants for arrays that are local or **common.** A formal argument array may have intervals that are of length equal to one of the other formal arguments. An element of an array is denoted by the array name followed by a parenthesized comma-separated list of integer values, each of which must lie within the corresponding interval. (The intervals may include negative numbers.) Entire arrays may be passed as procedure arguments or in input/output lists, or they may be initialized; all other array references must be to individual elements.

4.5. Structures

It is possible to define new types which are made up of elements of other types. The compound object is known as a *structure;* its constituents are called *members* of the structure. The structure may be given a name, which acts as a type name in the remaining statements within the scope of its declaration. The elements of a structure may be of any type (including previously defined structures), or they may be arrays of such objects. Entire structures may be passed to procedures or be used in input/output lists; individual elements of structures may be referenced. The uses of structures will be detailed below. The following structure might represent a symbol table:

```
struct tableentry
        {
        character(8) name
        integer hashvalue
        integer numberofelements
        field(0:1) initialized, used, set
        field(0:10) type
        }
```

5. EXPRESSIONS

Expressions are syntactic forms that yield a value. An expression may have any of the following forms, recursively applied:

primary
(*expression*)
unary-operator expression
expression binary-operator expression

In the following table of operators, all operators on a line have equal precedence and have higher precedence than operators on later lines. The meanings of these operators are described in sections 5.3 and 5.4.

```
-> .
**
* /     unary + - ++ --
+ -
< <= > >= == ~=
& &&
| ||
$
= += -= *= /= **=   &= |= &&= ||=
```

Examples of expressions are

```
a<b && b<c
-(a + sin(x)) / (5+cos(x))**2
```

5.1. Primaries

Primaries are the basic elements of expressions, as follows:

5.1.1. Constants

Constants are described in Section 4.2.

5.1.2. Variables

Scalar variable names are primaries. They may appear on the left or the right side of an assignment. Unqualified names of aggregates (structures or arrays) may only appear as procedure arguments and in input/output lists.

5.1.3. Array Elements

An element of an array is denoted by the array name followed by a parenthesized list of subscripts, one integer value for each declared dimension:

```
a(5)
b(6,-3,4)
```

5.1.4. Structure Members

A structure name followed by a dot followed by the name of a member of that structure constitutes a reference to that element. If that element is itself a structure, the reference may be further qualified.

```
a.b
x(3).y(4).z(5)
```

5.1.5. Procedure Invocations

A procedure is invoked by an expression of one of the forms

procedurename ( )
procedurename (*expression*)
procedurename (*expression-1*, ..., *expression-n*)

The *procedurename* is either the name of a variable declared **external** or it is the name of a function known to the EFL compiler (see Section 8.5), or it is the actual name of a procedure, as it appears in a **procedure** statement. If a *procedurename* is declared **external** and is an argument of the current procedure, it is associated with the procedure name passed as actual argument; otherwise it is the actual name of a procedure. Each *expression* in the above is called an *actual argument*. Examples of procedure invocations are

```
f(x)
work()
g(x, y+3, 'xx')
```

When one of these procedure invocations is to be performed, each of the actual argument expressions is first evaluated. The types, precisions, and bounds of actual and formal arguments should agree. If an actual argument is a variable name, array element, or structure member, the called procedure is permitted to use the corresponding formal argument as the left side of an assignment or in an input list; otherwise it may only use the value. After the formal and actual arguments are associated, control is passed to the first executable statement of the procedure. When a **return** statement is executed in that procedure, or when control reaches the **end** statement of that procedure, the function value is made available as the value of the procedure invocation. The type of the value is determined by the attributes of the *procedurename* that are declared or implied in the calling procedure, which must agree with the attributes declared for the function in its procedure. In the special case of a generic function, the type of the result is also affected by the type of the argument. See Chapter 8 for details.

5.1.6. Input/Output Expressions

The EFL input/output syntactic forms may be used as integer primaries that have a nonzero value if an error occurs during the input or output. See Section 7.7.

5.1.7. Coercions

An expression of one precision or type may be converted to another by an expression of the form

attributes (*expression*)

At present, the only *attributes* permitted are precision and basic types. Attributes are separated by white space. An arithmetic value of one type may be coerced to any other arithmetic type; a character expression of one length may be coerced to a character expression of another length; logical expressions may not be coerced to a nonlogical type. As a special case, a quantity of **complex** or **long complex** type may be constructed from two integer or real quantities by passing two expressions (separated by a comma) in the coercion. Examples and equivalent values are

integer(5.3) = **5**
long real(5) = **5.0d0**
complex(5,3) = $5+3i$

Most conversions are done implicitly, since most binary operators permit operands of different arithmetic types. Explicit coercions are of most use when it is necessary to convert the type of an actual argument to match that of the corresponding formal parameter in a procedure call.

5.1.8. Sizes

There is a notation which yields the amount of memory required to store a datum or an item of specified type:

sizeof (*leftside*)
sizeof (*attributes*)

In the first case, *leftside* can denote a variable, array, array element, or structure member. The value of **sizeof** is an integer, which gives the size in arbitrary units. If the size is needed in terms of the size of some specific unit, this can be computed by division:

sizeof(x) / sizeof(integer)

yields the size of the variable **x** in integer words.

The distance between consecutive elements of an array may not equal **sizeof** because certain data types require final padding on some machines. The **lengthof** operator gives this larger value, again in arbitrary units. The syntax is

lengthof (*leftside*)
lengthof (*attributes*)

5.2. Parentheses

An expression surrounded by parentheses is itself an expression. A parenthesized expression must be evaluated before an expression of which it is a part is evaluated.

5.3. Unary Operators

All of the unary operators in EFL are prefix operators. The result of a unary operator has the same type as its operand.

5.3.1. Arithmetic

Unary + has no effect. A unary – yields the negative of its operand.

The prefix operator ++ adds one to its operand. The prefix operator — subtracts one from its operand. The value of either expression is the result of the addition or subtraction. For these two operators, the operand must be a scalar, array element, or structure member of arithmetic type. (As a side effect, the operand value is changed.)

5.3.2. Logical

The only logical unary operator is complement (~). This operator is defined by the equations

~ true = false
~ false = true

5.4. Binary Operators

Most EFL operators have two operands, separated by the operator. Because the character set must be limited, some of the operators are denoted by strings of two or three special characters. All binary operators except exponentiation are left associative.

5.4.1. Arithmetic

The binary arithmetic operators are

+	addition
−	subtraction
*	multiplication
/	division
**	exponentiation

Exponentiation is right associative: a**b**c = a**(b**c) = $a^{(b^c)}$ The operations have the conventional meanings: 8+2 = 10, 8−2 = 6, 8*2 = 16, 8/2 = 4, 8**2 = 8^2 = 64.

The type of the result of a binary operation *A op B* is determined by the types of its operands:

	Type of B				
Type of A	integer	real	long real	complex	long complex
integer	integer	real	long real	complex	long complex
real	real	real	long real	complex	long complex
long real	long real	long real	long real	long complex	long complex
complex	complex	complex	long complex	complex	long complex
long complex	long complex	long complex	long complex	long complex	long complex

If the type of an operand differs from the type of the result, the calculation is done as if the operand were first coerced to the type of the result. If both operands are integers, the result is of type integer, and is computed exactly. (Quotients are truncated toward zero, so 8/3=2.)

5.4.2. Logical

The two binary logical operations in EFL, **and** and **or,** are defined by the truth tables:

A	B	A and B	A or B
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

Each of these operators comes in two forms. In one form, the order of evaluation is specified. The expression

a && b

is evaluated by first evaluating **a**; if it is false then the expression is false and **b** is not evaluated; otherwise the expression has the value of **b.** The expression

a || b

is evaluated by first evaluating **a;** if it is true then the expression is true and **b** is not evaluated;

otherwise the expression has the value of **b.** The other forms of the operators (**&** for **and** and **|** for **or**) do not imply an order of evaluation. With the latter operators, the compiler may speed up the code by evaluating the operands in any order.

5.4.3. Relational Operators

There are six relations between arithmetic quantities. These operators are not associative.

EFL Operator		Meaning
<	$<$	less than
<=	$\leq$	less than or equal to
==	$=$	equal to
~=	$\neq$	not equal to
>	$>$	greater than
>=	$\geq$	greater than or equal

Since the complex numbers are not ordered, the only relational operators that may take complex operands are == and ~= . The character collating sequence is not defined.

5.4.4. Assignment Operators

All of the assignment operators are right associative. The simple form of assignment is

basic-left-side = *expression*

A *basic-left-side* is a scalar variable name, array element, or structure member of basic type. This statement computes the expression on the right side, and stores that value (possibly after coercing the value to the type of the left side) in the location named by the left side. The value of the assignment expression is the value assigned to the left side after coercion.

There is also an assignment operator corresponding to each binary arithmetic and logical operator. In each case, *a op= b* is equivalent to *a = a op b*. (The operator and equal sign must not be separated by blanks.) Thus, **n+=2** adds 2 to n. The location of the left side is evaluated only once.

5.5. Dynamic Structures

EFL does not have an address (pointer, reference) type. However, there is a notation for dynamic structures,

leftside -> *structurename*

This expression is a structure with the shape implied by *structurename* but starting at the location of *leftside*. In effect, this overlays the structure template at the specified location. The *leftside* must be a variable, array, array element, or structure member. The type of the *leftside* must be one of the types in the structure declaration. An element of such a structure is denoted in the usual way using the dot operator. Thus,

place(i) -> st.elt

refers to the **elt** member of the **st** structure starting at the i^{th} element of the array **place.**

5.6. Repetition Operator

Inside of a list, an element of the form

integer-constant-expression **$** *constant-expression*

is equivalent to the appearance of the *expression* a number of times equal to the first expression. Thus,

(3, 3$4, 5)

is equivalent to

(3, 4, 4, 4, 5)

5.7. Constant Expressions

If an expression is built up out of operators (other than functions) and constants, the value of the expression is a constant, and may be used anywhere a constant is required.

6. DECLARATIONS

Declarations statement describe the meaning, shape, and size of named objects in the EFL language.

6.1. Syntax

A declaration statement is made up of attributes and variables. Declaration statements are of two form:

attributes variable-list
attributes { declarations }

In the first case, each name in the *variable-list* has the specified attributes. In the second, each name in the declarations also has the specified attributes. A variable name may appear in more than one variable list, so long as the attributes are not contradictory. Each name of a nonargument variable may be accompanied by an initial value specification. The *declarations* inside the braces are one or more declaration statements. Examples of declarations are

```
integer k=2

long real b(7,3)

common(cname)
    {
    integer i
    long real array(5,0:3) x, y
    character(7) ch
    }
```

6.2. Attributes

6.2.1. Basic Types

The following are basic types in declarations

logical
integer
field(m:n)
character(k)
real
complex

In the above, the quantities k, m, and n denote integer constant expressions with the properties $k>0$ and $n>m$.

6.2.2. Arrays

The dimensionality may be declared by an **array** attribute

array(b_1,...,b_n)

Each of the b_i may either be a single integer expression or a pair of integer expressions separated by a colon. The pair of expressions form a lower and an upper bound; the single expression is an

upper bound with an implied lower bound of 1. The number of dimensions is equal to n, the number of bounds. All of the integer expressions must be constants. An exception is permitted only if all of the variables associated with an array declarator are formal arguments of the procedure; in this case, each bound must have the property that *upper−lower+1* is equal to a formal argument of the procedure. (The compiler has limited ability to simplify expressions, but it will recognize important cases such as **(0:n−1)**. The upper bound for the last dimension (b_n) may be marked by an asterisk (*) if the size of the array is not known. The following are legal **array** attributes:

```
array(5)
array(5, 1:5, -3:0)
array(5, *)
array(0:m-1, m)
```

6.2.3. Structures

A structure declaration is of the form

struct *structname* { *declaration statements* }

The *structname* is optional; if it is present, it acts as if it were the name of a type in the rest of its scope. Each name that appears inside the *declarations* is a *member* of the structure, and has a special meaning when used to qualify any variable declared with the structure type. A name may appear as a member of any number of structures, and may also be the name of an ordinary variable, since a structure member name is used only in contexts where the parent type is known. The following are valid structure attributes

```
struct xx
    {
    integer a, b
    real x(5)
    }

struct { xx z(3); character(5) y }
```

The last line defines a structure containing an array of three **xx**′ *s* and a character string.

6.2.4. Precision

Variables of floating point (**real** or **complex**) type may be declared to be **long** to ensure they have higher precision than ordinary floating point variables. The default precision is **short**.

6.2.5. Common

Certain objects called *common areas* have external scope, and may be referenced by any procedure that has a declaration for the name using a

common (*commonareaname*)

attribute. All of the variables declared with a particular **common** attribute are in the same block; the order in which they are declared is significant. Declarations for the same block in differing procedures must have the variables in the same order and with the same types, precision, and shapes, though not necessarily with the same names.

6.2.6. External

If a name is used as the procedure name in a procedure invocation, it is implicitly declared to have the **external** attribute. If a procedure name is to be passed as an argument, it is necessary to declare it in a statement of the form

external [[*name*]]

If a name has the external attribute and it is a formal argument of the procedure, then it is associated with a procedure identifier passed as an actual argument at each call. If the name is not a formal argument, then that name is the actual name of a procedure, as it appears in the corresponding **procedure** statement.

6.3. Variable List

The elements of a variable list in a declaration consist of a name, an optional dimension specification, and an optional initial value specification. The name follows the usual rules. The dimension specification is the same form and meaning as the parenthesized list in an **array** attribute. The initial value specification is an equal sign (=) followed by a constant expression. If the name is an array, the right side of the equal sign may be a parenthesized list of constant expressions, or repeated elements or lists; the total number of elements in the list must not exceed the number of elements of the array, which are filled in column-major order.

6.4. The Initial Statement

An initial value may also be specified for a simple variable, array, array element, or member of a structure using a statement of the form

initial [[*var* = *val*]]

The *var* may be a variable name, array element specification, or member of structure. The right side follows the same rules as for an initial value specification in other declaration statements.

7. EXECUTABLE STATEMENTS

Every useful EFL program contains executable statements — otherwise it would not do anything and would not need to be run. Statements are frequently made up of other statements. Blocks are the most obvious case, but many other forms contain statements as constituents.

To increase the legibility of EFL programs, some of the statement forms can be broken without an explicit continuation. A square (□) in the syntax represents a point where the end of a line will be ignored.

7.1. Expression Statements

7.1.1. Subroutine Call

A procedure invocation that returns no value is known as a subroutine call. Such an invocation is a statement. Examples are

```
work(in, out)
run( )
```

Input/output statements (see Section 7.7) resemble procedure invocations but do not yield a value. If an error occurs the program stops.

7.1.2. Assignment Statements

An expression that is a simple assignment (=) or a compound assignment (+= etc.) is a statement:

```
a = b
a = sin(x)/6
x *= y
```

7.2. Blocks

A block is a compound statement that acts as a statement. A block begins with a left brace, optionally followed by declarations, optionally followed by executable statements, followed by a right brace. A block may be used anywhere a statement is permitted. A block is not an expression and does not have a value. An example of a block is

```
{
integer i    # this variable is unknown outside the braces
big = 0
do i = 1,n
   if(big < a(i))
        big = a(i)
}
```

7.3. Test Statements

Test statements permit execution of certain statements conditional on the truth of a predicate.

7.3.1. If Statement

The simplest of the test statements is the **if** statement, of form

if (*logical-expression*) □ *statement*

The logical expression is evaluated; if it is true, then the *statement* is executed.

7.3.2. If-Else

A more general statement is of the form

if (*logical-expression*) □ *statement-1* □ **else** □ *statement-2*

If the expression is **true** then *statement-1* is executed, otherwise *statement-2* is executed. Either of the consequent statements may itself be an **if-else** so a completely nested test sequence is possible:

```
if(x<y)
   if(a<b)
      k = 1
   else
      k = 2
else
   if(a<b)
      m = 1
   else
      m = 2
```

An **else** applies to the nearest preceding un-**else**d **if**. A more common use is as a sequential test:

```
if(x==1)
   k = 1
else if(x==3 | x==5)
   k = 2
else
   k = 3
```

7.3.3. Select Statement

A multiway test on the value of a quantity is succinctly stated as a **select** statement, which has the general form

select(*expression* **)** □ *block*

Inside the block two special types of labels are recognized. A prefix of the form

case ⟦ *constant* ⟧ **:**

marks the statement to which control is passed if the expression in the select has a value equal to one of the case constants. If the expression equals none of these constants, but there is a label **default** inside the select, a branch is taken to that point; otherwise the statement following the right brace is executed. Once execution begins at a **case** or **default** label, it continues until the next **case** or **default** is encountered. The **else-if** example above is better written as

```
select(x)
   {
   case 1:
      k = 1
   case 3,5:
      k = 2
   default:
      k = 3
   }
```

Note that control does not 'fall through' to the next case.

7.4. Loops

The loop forms provide the best way of repeating a statement or sequence of operations. The simplest (**while**) form is theoretically sufficient, but it is very convenient to have the more general loops available, since each expresses a mode of control that arises frequently in practice.

7.4.1. While Statement

This construct has the form

while (*logical-expression* **)** □ *statement*

The expression is evaluated; if it is true, the statement is executed, and then the test is performed again. If the expression is false, execution proceeds to the next statement.

7.5. For Statement

The **for** statement is a more elaborate looping construct. It has the form

for (*initial-statement* **,** □ *logical-expression* **,** □ *iteration-statement* **)** □ *body-statement*

Except for the behavior of the **next** statement (see Section 7.6.3), this construct is equivalent to

```
initial-statement
while ( logical-expression )
   {
   body-statement
   iteration-statement
   }
```

This form is useful for general arithmetic iterations, and for various pointer-type operations. The sum of the integers from 1 to 100 can be computed by the fragment

```
n = 0
for(i = 1, i <= 100, i += 1)
    n += i
```

Alternatively, the computation could be done by the single statement

```
for( { n = 0 ; i = 1 } , i<=100 , { n += i ; ++i } )
    ;
```

Note that the body of the **for** loop is a null statement in this case. An example of following a linked list will be given later.

7.5.1. Repeat Statement

The statement

repeat □ *statement*

executes the statement, then does it again, without any termination test. Obviously, a test inside the *statement* is needed to stop the loop.

7.5.2. Repeat...Until Statement

The **while** loop performs a test before each iteration. The statement

repeat □ *statement* □ **until** (*logical-expression*)

executes the *statement,* then evaluates the logical; if the logical is true the loop is complete; otherwise control returns to the *statement.* Thus, the body is always executed at least once. The **until** refers to the nearest preceding **repeat** that has not been paired with an **until.** In practice, this appears to be the least frequently used looping construct.

7.5.3. Do Loops

The simple arithmetic progression is a very common one in numerical applications. EFL has a special loop form for ranging over an ascending arithmetic sequence

do *variable* = *expression-1, expression-2, expression-3*
    statement

The variable is first given the value *expression-1.* The statement is executed, then *expression-3* is added to the variable. The loop is repeated until the variable exceeds *expression-2.* If *expression-3* and the preceding comma are omitted, the increment is taken to be 1. The loop above is equivalent to

```
t2 = expression-2
t3 = expression-3
for(variable = expression-1 , variable <= t2 , variable += t3)
    statement
```

(The compiler translates EFL **do** statements into Fortran DO statements, which are in turn usually compiled into excellent code.) The **do** *variable* may not be changed inside of the loop, and *expression-1* must not exceed *expression-2.* The sum of the first hundred positive integers could be computed by

```
n = 0
do i = 1, 100
        n += i
```

7.6. Branch Statements

Most of the need for branch statements in programs can be averted by using the loop and test constructs, but there are programs where they are very useful.

7.6.1. Goto Statement

The most general, and most dangerous, branching statement is the simple unconditional

goto *label*

After executing this statement, the next statement performed is the one following the given label. Inside of a **select** the case labels of that block may be used as labels, as in the following example:

```
select(k)
        {
        case 1:
                error(7)

        case 2:
                k = 2
                goto case 4

        case 3:
                k = 5
                goto case 4

        case 4:
                fixup(k)
                goto default

        default:
                prmsg("ouch")
        }
```

(If two **select** statements are nested, the case labels of the outer **select** are not accessible from the inner one.)

7.6.2. Break Statement

A safer statement is one which transfers control to the statement following the current **select** or loop form. A statement of this sort is almost always needed in a **repeat** loop:

```
repeat
    {
    do a computation
    if( finished )
        break
    }
```

More general forms permit controlling a branch out of more than one construct.

break 3

transfers control to the statement following the third loop and/or **select** surrounding the statement. It is possible to specify which type of construct (**for**, **while**, **repeat**, **do**, or **select**) is to be counted. The statement

break while

breaks out of the first surrounding **while** statement. Either of the statements

break 3 for
break for 3

will transfer to the statement after the third enclosing **for** loop.

7.6.3. Next Statement

The **next** statement causes the first surrounding loop statement to go on to the next iteration: the next operation performed is the test of a **while,** the *iteration-statement* of a **for,** the body of a **repeat,** the test of a **repeat...until,** or the increment of a **do.** Elaborations similar to those for **break** are available:

next
next 3
next 3 for
next for 3

A **next** statement ignores **select** statements.

7.6.4. Return

The last statement of a procedure is followed by a return of control to the caller. If it is desired to effect such a return from any other point in the procedure, a

return

statement may be executed. Inside a function procedure, the function value is specified as an argument of the statement:

return (*expression*)

7.7. Input/Output Statements

EFL has two input statements (**read** and **readbin**), two output statements (**write** and **writebin**), and three control statements (**endfile**, **rewind**, and **backspace**). These forms may be used either as a primary with a **integer** value or as a statement. If an exception occurs when one of these forms is used as a statement, the result is undefined but will probably be treated as a fatal error. If they are used in a context where they return a value, they return zero if no exception occurs. For the input forms, a negative value indicates end-of-file and a positive value an error. The input/output part of EFL very strongly reflects the facilities of Fortran.

7.7.1. Input/Output Units

Each I/O statement refers to a 'unit', identified by a small positive integer. Two special units are defined by EFL, the *standard input unit* and the *standard output unit.* These particular units are assumed if no unit is specified in an I/O transmission statement.

The data on the unit are organized into *records.* These records may be read or written in a fixed sequence, and each transmission moves an integral number of records. Transmission proceeds from the first record until the *end of file.*

7.7.2. Binary Input/Output

The **readbin** and **writebin** statements transmit data in a machine-dependent but swift manner. The statements are of the form

writebin(*unit* **,** *binary-output-list* **)**
readbin(*unit* **,** *binary-input-list* **)**

Each statement moves one unformatted record between storage and the device. The *unit* is an integer expression. A *binary-output-list* is an *iolist* (see below) without any format specifiers. A *binary-input-list* is an *iolist* without format specifiers in which each of the expressions is a variable

name, array element, or structure member.

7.7.3. Formatted Input/Output

The **read** and **write** statements transmit data in the form of lines of characters. Each statement moves one or more records (lines). Numbers are translated into decimal notation. The exact form of the lines is determined by format specifications, whether provided explicitly in the statement or implicitly. The syntax of the statements is

write(*unit* **,** *formatted-output-list* **)**
read(*unit* **,** *formatted-input-list* **)**

The lists are of the same form as for binary I/O, except that the lists may include format specifications. If the *unit* is omitted, the standard input or output unit is used.

7.7.4. Iolists

An *iolist* specifies a set of values to be written or a set of variables into which values are to be read. An *iolist* is a list of one or more *ioexpressions* of the form

expression
{ *iolist* }
do-specification { *iolist* }

For formatted I/O, an *ioexpression* may also have the forms

ioexpression : *format-specifier*
: *format-specifier*

A *do-specification* looks just like a **do** statement, and has a similar effect: the values in the braces are transmitted repeatedly until the **do** execution is complete.

7.7.5. Formats

The following are permissible *format-specifiers*. The quantities w, d, and k must be integer constant expressions.

i(w)	integer with w digits
f(w,d)	floating point number of w characters, d of them to the right of the decimal point.
e(w,d)	floating point number of w characters, d of them to the right of the decimal point, with the exponent field marked with the letter **e**
l(w)	logical field of width w characters, the first of which is **t** or **f** (the rest are blank on output, ignored on input) standing for **true** and **false** respectively
c	character string of width equal to the length of the datum
c(w)	character string of width w
s(k)	skip k lines
x(k)	skip k spaces
" ... "	use the characters inside the string as a Fortran format

If no format is specified for an item in a formatted input/output statement, a default form is chosen.

If an item in a list is an array name, then the entire array is transmitted as a sequence of elements, each with its own format. The elements are transmitted in column-major order, the same order used for array initializations.

7.7.6. Manipulation statements

The three input/output statements

backspace(*unit***)**
rewind(*unit***)**
endfile(*unit***)**

look like ordinary procedure calls, but may be used either as statements or as integer expressions which yield non-zero if an error is detected. **backspace** causes the specified unit to back up, so that the next read will re-read the previous record, and the next write will over-write it. **rewind** moves the device to its beginning, so that the next input statement will read the first record. **endfile** causes the file to be marked so that the record most recently written will be the last record on the file, and any attempt to read past is an error.

8. PROCEDURES

Procedures are the basic unit of an EFL program, and provide the means of segmenting a program into separately compilable and named parts.

8.1. Procedure Statement

Each procedure begins with a statement of one of the forms

procedure
attributes **procedure** *procedurename*
attributes **procedure** *procedurename* ( )
attributes **procedure** *procedurename* ([[*name*]])

The first case specifies the main procedure, where execution begins. In the two other cases, the *attributes* may specify precision and type, or they may be omitted entirely. The precision and type of the procedure may be declared in an ordinary declaration statement. If no type is declared, then the procedure is called a *subroutine* and no value may be returned for it. Otherwise, the procedure is a function and a value of the declared type is returned for each call. Each *name* inside the parentheses in the last form above is called a *formal argument* of the procedure.

8.2. End Statement

Each procedure terminates with a statement

end

8.3. Argument Association

When a procedure is invoked, the actual arguments are evaluated. If an actual argument is the name of a variable, an array element, or a structure member, that entity becomes associated with the formal argument, and the procedure may reference the values in the object, and assign to it. Otherwise, the value of the actual is associated with the formal argument, but the procedure may not attempt to change the value of that formal argument.

If the value of one of the arguments is changed in the procedure, it is not permitted that the corresponding actual argument be associated with another formal argument or with a **common** element that is referenced in the procedure.

8.4. Execution and Return Values

After actual and formal arguments have been associated, control passes to the first executable statement of the procedure. Control returns to the invoker either when the **end** statement of the procedure is reached or when a **return** statement is executed. If the procedure is a function (has a declared type), and a **return(***value***)** is executed, the value is coerced to the correct type and precision and returned.

8.5. Known Functions

A number of functions are known to EFL, and need not be declared. The compiler knows the types of these functions. Some of them are *generic;* i.e., they name a family of functions that differ in the types of their arguments and return values. The compiler chooses which element of the set to invoke based upon the attributes of the actual arguments.

8.5.1. Minimum and Maximum Functions

The generic functions are **min** and **max.** The **min** calls return the value of their smallest argument; the **max** calls return the value of their largest argument. These are the only functions that may take different numbers of arguments in different calls. If any of the arguments are **long real** then the result is **long real**. Otherwise, if any of the arguments are **real** then the result is **real;** otherwise all the arguments and the result must be **integer.** Examples are

```
min(5, x, -3.20)
max(i, z)
```

8.5.2. Absolute Value

The **abs** function is a generic function that returns the magnitude of its argument. For integer and real arguments the type of the result is identical to the type of the argument; for complex arguments the type of the result is the real of the same precision.

8.5.3. Elementary Functions

The following generic functions take arguments of **real**, **long real**, or **complex** type and return a result of the same type:

sin	**sine function**
cos	**cosine function**
exp	**exponential function (e^x).**
log	**natural (base e) logarithm**
log10	**common (base 10) logarithm**
sqrt	**square root function ($\sqrt{x}$).**

In addition, the following functions accept only **real** or **long real** arguments:

atan	$atan(x)=\tan^{-1}x$
atan2	$atan2(x,y)=\tan^{-1}\frac{x}{y}$

8.5.4. Other Generic Functions

The **sign** functions takes two arguments of identical type; **sign**$(x,y) = sgn(y)\,|x\,|$. The **mod** function yields the remainder of its first argument when divided by its second. These functions accept integer and real arguments.

9. ATAVISMS

Certain facilities are included in the EFL language to ease the conversion of old Fortran or Ratfor programs to EFL.

9.1. Escape Lines

In order to make use of nonstandard features of the local Fortran compiler, it is occasionally necessary to pass a particular line through to the EFL compiler output. A line that begins with a percent sign ('%') is copied through to the output, with the percent sign removed but no other change. Inside of a procedure, each escape line is treated as an executable statement. If a sequence

of lines constitute a continued Fortran statement, they should be enclosed in braces.

9.2. Call Statement

A subroutine call may be preceded by the keyword **call.**

```
call joe
call work(17)
```

9.3. Obsolete Keywords

The following keywords are recognized as synonyms of EFL keywords:

Fortran	EFL
double precision	**long real**
function	**procedure**
subroutine	**procedure** *(untyped)*

9.4. Numeric Labels

Standard statement labels are identifiers. A numeric (positive integer constant) label is also permitted; the colon is optional following a numeric label.

9.5. Implicit Declarations

If a name is used but does not appear in a declaration, the EFL compiler gives a warning and assumes a declaration for it. If it is used in the context of a procedure invocation, it is assumed to be a procedure name; otherwise it is assumed to be a local variable defined at nesting level 1 in the current procedure. The assumed type is determined by the first letter of the name. The association of letters and types may be given in an **implicit** statement, with syntax

implicit (*letter-list*) *type*

where a *letter-list* is a list of individual letters or ranges (pair of letters separated by a minus sign). If no **implicit** statement appears, the following rules are assumed:

```
implicit (a–h, o–z) real
implicit (i–n) integer
```

9.6. Computed goto

Fortran contains an indexed multi-way branch; this facility may be used in EFL by the computed GOTO:

goto ([[*label*]]), *expression*

The expression must be of type integer and be positive but be no larger than the number of labels in the list. Control is passed to the statement marked by the label whose position in the list is equal to the expression.

9.7. Go To Statement

In unconditional and computed **goto** statements, it is permissible to separate the **go** and **to** words, as in

```
go to xyz
```

9.8. Dot Names

Fortran uses a restricted character set, and represents certain operators by multi-character sequences. There is an option (**dots=on**; see Section 10.2) which forces the compiler to recognize the forms in the second column below:

<	**.lt.**
<=	**.le.**
>	**.gt.**
>=	**.ge.**
==	**.eq.**
~=	**.ne.**
&	**.and.**
|	**.or.**
&&	**.andand.**
||	**.oror.**
~	**.not.**
true	**.true.**
false	**.false.**

In this mode, no structure element may be named **lt, le,** etc. The readable forms in the left column are always recognized.

9.9. Complex Constants

A complex constant may be written as a parenthesized list of real quantities, such as

(1.5, 3.0)

The preferred notation is by a type coercion,

complex(1.5, 3.0)

9.10. Function Values

The preferred way to return a value from a function in EFL is the **return**(*value*) construct. However, the name of the function acts as a variable to which values may be assigned; an ordinary **return** statement returns the last value assigned to that name as the function value.

9.11. Equivalence

A statement of the form

equivalence v_1, v_2, ..., v_n

declares that each of the v_i starts at the same memory location. Each of the v_i may be a variable name, array element name, or structure member.

9.12. Minimum and Maximum Functions

There are a number of non-generic functions in this category, which differ in the required types of the arguments and the type of the return value. They may also have variable numbers of arguments, but all the arguments must have the same type.

Function	Argument Type	Result Type
amin0	**integer**	**real**
amin1	**real**	**real**
min0	**integer**	**integer**
min1	**real**	**integer**
dmin1	**long real**	**long real**
amax0	**integer**	**real**
amax1	**real**	**real**
max0	**integer**	**integer**
max1	**real**	**integer**
dmax1	**long real**	**long real**

10. COMPILER OPTIONS

A number of options can be used to control the output and to tailor it for various compilers and systems. The defaults chosen are conservative, but it is sometimes necessary to change the output to match peculiarities of the target environment.

Options are set with statements of the form

option [[*opt*]]

where each *opt* is of one of the forms

optionname
optionname = *optionvalue*

The *optionvalue* is either a constant (numeric or string) or a name associated with that option. The two names **yes** and **no** apply to a number of options.

10.1. Default Options

Each option has a default setting. It is possible to change the whole set of defaults to those appropriate for a particular environment by using the **system** option. At present, the only valid values are **system=unix** and **system=gcos.**

10.2. Input Language Options

The **dots** option determines whether the compiler recognizes **.lt.** and similar forms. The default setting is **no.**

10.3. Input/Output Error Handling

The **ioerror** option can be given three values: **none** means that none of the I/O statements may be used in expressions, since there is no way to detect errors. The implementation of the **ibm** form uses ERR= and END= clauses. The implementation of the **fortran77** form uses IOSTAT= clauses.

10.4. Continuation Conventions

By default, continued Fortran statements are indicated by a character in column 6 (Standard Fortran). The option **continue=column1** puts an ampersand (**&**) in the first column of the continued lines instead.

10.5. Default Formats

If no format is specified for a datum in an iolist for a **read** or **write** statement, a default is provided. The default formats can be changed by setting certain options

Option	Type
iformat	integer
rformat	real
dformat	long real
zformat	complex
zdformat	long complex
lformat	logical

The associated value must be a Fortran format, such as

option rformat=f22.6

10.6. Alignments and Sizes

In order to implement **character** variables, structures, and the **sizeof** and **lengthof** operators, it is necessary to know how much space various Fortran data types require, and what boundary alignment properties they demand. The relevant options are

Fortran Type	Size Option	Alignment Option
integer	**isize**	**ialign**
real	**rsize**	**ralign**
long real	**dsize**	**dalign**
complex	**zsize**	**zalign**
logical	**lsize**	**lalign**

The sizes are given in terms of an arbitrary unit; the alignment is given in the same units. The option **charperint** gives the number of characters per **integer** variable.

10.7. Default Input/Output Units

The options **ftnin** and **ftnout** are the numbers of the standard input and output units. The default values are **ftnin=5** and **ftnout=6.**

10.8. Miscellaneous Output Control Options

Each Fortran procedure generated by the compiler will be preceded by the value of the **procheader** option.

No Hollerith strings will be passed as subroutine arguments if **hollincall=no** is specified.

The Fortran statement numbers normally start at 1 and increase by 1. It is possible to change the increment value by using the **deltastno** option.

11. EXAMPLES

In order to show the flavor or programming in EFL, we present a few examples. They are short, but show some of the convenience of the language.

11.1. File Copying

The following short program copies the standard input to the standard output, provided that the input is a formatted file containing lines no longer than a hundred characters.

```
procedure  # main program
character(100) line

while( read( , line) == 0 )
      write( , line)
end
```

Since **read** returns zero until the end of file (or a read error), this program keeps reading and writing until the input is exhausted.

11.2. Matrix Multiplication

The following procedure multiplies the $m \times n$ matrix a by the $n \times p$ matrix b to give the $m \times p$ matrix c. The calculation obeys the formula $c_{ij} = \sum a_{ik} b_{kj}$.

```
procedure matmul(a,b,c, m,n,p)
integer i, j, k, m, n, p
long real a(m,n), b(n,p), c(m,p)
do i = 1,m
do j = 1,p
          {
          c(i,j) = 0
          do k = 1,n
                    c(i,j) += a(i,k) * b(k,j)
          }
end
```

11.3. Searching a Linked List

Assume we have a list of pairs of numbers (x,y). The list is stored as a linked list sorted in ascending order of x values. The following procedure searches this list for a particular value of x and returns the corresponding y value.

```
define LAST          0
define NOTFOUND -1

integer procedure val(list, first, x)

#  list is an array of structures.
#  Each structure contains a thread index value, an x, and a y value.
struct
          {
          integer nextindex
          integer x, y
          } list(*)
integer first, p, arg

for(p = first , p~=LAST && list(p).x<=x , p = list(p).nextindex)
          if(list(p).x == x)
                    return( list(p).y )

return(NOTFOUND)
end
```

The search is a single **for** loop that begins with the head of the list and examines items until either the list is exhausted (p==LAST) or until it is known that the specified value is not on the list

(list(p).x > x). The two tests in the conjunction must be performed in the specified order to avoid using an invalid subscript in the **list(p)** reference. Therefore, the **&&** operator is used. The next element in the chain is found by the iteration statement **p=list(p).nextindex**.

11.4. Walking a Tree

As an example of a more complicated problem, let us imagine we have an expression tree stored in a common area, and that we want to print out an infix form of the tree. Each node is either a leaf (containing a numeric value) or it is a binary operator, pointing to a left and a right descendant. In a recursive language, such a tree walk would be implement by the following simple pseudocode:

```
if this node is a leaf
        print its value
otherwise
        print a left parenthesis
        print the left node
        print the operator
        print the right node
        print a right parenthesis
```

In a nonrecursive language like EFL, it is necessary to maintain an explicit stack to keep track of the current state of the computation. The following procedure calls a procedure **outch** to print a single character and a procedure **outval** to print a value.

```
procedure walk(first)           # print out an expression tree

integer first                   # index of root node
integer currentnode
integer stackdepth
common(nodes) struct
            {
            character(1) op
            integer leftp, rightp
            real val
            } tree(100)         # array of structures

struct
            {
            integer nextstate
            integer nodep
            } stackframe(100)

define NODE         tree(currentnode)
define STACK        stackframe(stackdepth)

#   nextstate values
define DOWN         1
define LEFT         2
define RIGHT        3

#   initialize stack with root node
stackdepth = 1
STACK.nextstate = DOWN
STACK.nodep = first
```

```
while( stackdepth > 0 )
    {
    currentnode = STACK.nodep
    select(STACK.nextstate)
        {
        case DOWN:
            if(NODE.op == " ")  # a leaf
                {
                outval( NODE.val )
                stackdepth -= 1
                }
            else {  # a binary operator node
                outch( "(" )
                STACK.nextstate = LEFT
                stackdepth += 1
                STACK.nextstate = DOWN
                STACK.nodep = NODE.leftp
                }

        case LEFT:
            outch( NODE.op )
            STACK.nextstate = RIGHT
            stackdepth += 1
            STACK.nextstate = DOWN
            STACK.nodep = NODE.rightp

        case RIGHT:
            outch( ")" )
            stackdepth -= 1
        }
    }
end
```

12. PORTABILITY

One of the major goals of the EFL language is to make it easy to write portable programs. The output of the EFL compiler is intended to be acceptable to any Standard Fortran compiler (unless the **fortran77** option is specified).

12.1. Primitives

Certain EFL operations cannot be implemented in portable Fortran, so a few machine-dependent procedures must be provided in each environment.

12.1.1. Character String Copying

The subroutine **eflasc** is called to copy one character string to another. If the target string is shorter than the source, the final characters are not copied. If the target string is longer, its end is padded with blanks. The calling sequence is

```
subroutine eflasc(a, la, b, lb)
integer a(*), la, b(*), lb
```

and it must copy the first **lb** characters from **b** to the first **la** characters of **a**.

12.1.2. Character String Comparisons

The function **eflcmc** is invoked to determine the order of two character strings. The declaration is

```
integer function eflcmc(a, la, b, lb)
integer a(*), la, b(*), lb
```

The function returns a negative value if the string **a** of length **la** precedes the string **b** of length **lb**. It returns zero if the strings are equal, and a positive value otherwise. If the strings are of differing length, the comparison is carried out as if the end of the shorter string were padded with blanks.

13. ACKNOWLEDGMENTS

A. D. Hall originated the EFL language and wrote the first compiler for it; he also gave inestimable aid when I took up the project. B. W. Kernighan and W. S. Brown made a number of useful suggestions about the language and about this report. N. L. Schryer has acted as willing, cheerful, and severe first user and helpful critic of each new version and facility. J. L. Blue, L. C. Kaufman, and D. D. Warner made very useful contributions by making serious use of the compiler, and noting and tolerating its misbehaviors.

14. REFERENCE

1. B. W. Kernighan, "Ratfor — A Preprocessor for a Rational Fortran", Bell Laboratories Computing Science Technical Report #55

APPENDIX A. Relation Between EFL and Ratfor

There are a number of differences between Ratfor and EFL, since EFL is a defined language while Ratfor is the union of the special control structures and the language accepted by the underlying Fortran compiler. Ratfor running over Standard Fortran is almost a subset of EFL. Most of the features described in the Atavisms section are present to ease the conversion of Ratfor programs to EFL.

There are a few incompatibilities: The syntax of the **for** statement is slightly different in the two languages: the three clauses are separated by semicolons in Ratfor, but by commas in EFL. (The initial and iteration statements may be compound statements in EFL because of this change). The input/output syntax is quite different in the two languages, and there is no FORMAT statement in EFL. There are no ASSIGN or assigned GOTO statements in EFL.

The major linguistic additions are character data, factored declaration syntax, block structure, assignment and sequential test operators, generic functions, and data structures. EFL permits more general forms for expressions, and provides a more uniform syntax. (One need not worry about the Fortran/Ratfor restrictions on subscript or DO expression forms, for example.)

APPENDIX B. COMPILER

B.1. Current Version

The current version of the EFL compiler is a two-pass translator written in portable C. It implements all of the features of the language described above except for **long complex** numbers. Versions of this compiler run under the and UNIX† operating systems.

B.2. Diagnostics

The EFL compiler diagnoses all syntax errors. It gives the line and file name (if known) on which the error was detected. Warnings are given for variables that are used but not explicitly declared.

B.3. Quality of Fortran Produced

The Fortran produced by EFL is quite clean and readable. To the extent possible, the variable names that appear in the EFL program are used in the Fortran code. The bodies of loops and test constructs are indented. Statement numbers are consecutive. Few unneeded GOTO and CONTINUE statements are used. It is considered a compiler bug if incorrect Fortran is produced (except for escaped lines). The following is the Fortran procedure produced by the EFL compiler for the matrix multiplication example (Section 11.2):

```
      subroutine matmul(a, b, c, m, n, p)
      integer m, n, p
      double precision a(m, n), b(n, p), c(m, p)
      integer i, j, k
      do  3 i = 1, m
         do  2 j = 1, p
            c(i, j) = 0
            do  1 k = 1, n
               c(i, j) = c(i, j)+a(i, k)*b(k, j)
1              continue
2           continue
3        continue
      end
```

The following is the procedure for the tree walk (Section 11.4):

† UNIX is a trademark of Bell Laboratories.

```
      subroutine walk(first)
      integer first
      common /nodes/ tree
      integer tree(4, 100)
      real tree1(4, 100)
      integer staame(2, 100), stapth, curode
      integer const1(1)
      equivalence (tree(1,1), tree1(1,1))
      data const1(1)/4h     /
c print out an expression tree
c index of root node
c array of structures
c   nextstate values
c   initialize stack with root node
      stapth = 1
      staame(1, stapth) = 1
      staame(2, stapth) = first
   1  if (stapth .le. 0) goto  9
         curode = staame(2, stapth)
         goto  7
   2        if (tree(1, curode) .ne. const1(1)) goto 3
               call outval(tree1(4, curode))
c a leaf
               stapth = stapth-1
               goto  4
   3           call outch(1h()
c a binary operator node
               staame(1, stapth) = 2
               stapth = stapth+1
               staame(1, stapth) = 1
               staame(2, stapth) = tree(2, curode)
   4        goto  8
   5        call outch(tree(1, curode))
            staame(1, stapth) = 3
            stapth = stapth+1
            staame(1, stapth) = 1
            staame(2, stapth) = tree(3, curode)
            goto  8
   6        call outch(1h))
            stapth = stapth-1
            goto  8
   7        if (staame(1, stapth) .eq. 3) goto  6
            if (staame(1, stapth) .eq. 2) goto  5
            if (staame(1, stapth) .eq. 1) goto  2
   8     continue
         goto  1
   9  continue
      end
```

APPENDIX C. CONSTRAINTS ON THE DESIGN OF THE EFL LANGUAGE

Although Fortran can be used to simulate any finite computation, there are realistic limits on the generality of a language that can be translated into Fortran. The design of EFL was constrained by the implementation strategy. Certain of the restrictions are petty (six character

external names), but others are sweeping (lack of pointer variables). The following paragraphs describe the major limitations imposed by Fortran.

C.1. External Names

External names (procedure and COMMON block names) must be no longer than six characters in Fortran. Further, an external name is global to the entire program. Therefore, EFL can support block structure within a procedure, but it can have only one level of external name if the EFL procedures are to be compilable separately, as are Fortran procedures.

C.2. Procedure Interface

The Fortran standards, in effect, permit arguments to be passed between Fortran procedures either by reference or by copy-in/copy-out. This indeterminacy of specification shows through into EFL. A program that depends on the method of argument transmission is illegal in either language.

There are no procedure-valued variables in Fortran: a procedure name may only be passed as an argument or be invoked; it cannot be stored. Fortran (and EFL) would be noticeably simpler if a procedure variable mechanism were available.

C.3. Pointers

The most grievous problem with Fortran is its lack of a pointer-like data type. The implementation of the compiler would have been far easier if certain hard cases could have been handled by pointers. Further, the language could have been simplified considerably if pointers were accessible in Fortran. (There are several ways of simulating pointers by using subscripts, but they founder on the problems of external variables and initialization.)

C.4. Recursion

Fortran procedures are not recursive, so it was not practical to permit EFL procedures to be recursive. (Recursive procedures with arguments can be simulated only with great pain.)

C.5. Storage Allocation

The definition of Fortran does not specify the lifetime of variables. It would be possible but cumbersome to implement stack or heap storage disciplines by using COMMON blocks.

Berkeley FP User's Manual, Rev. 4.1

by

Scott Baden

ABSTRACT

This manual describes the Berkeley implementation of Backus' Functional Programming Language, FP. Since this implementation differs from Backus' original description of the language, those familiar with the literature will need to read about the system commands and the local modifications.

January 12, 1987

Table of Contents

1. Background

FP stands for a *Functional Programming* language. Functional programs deal with *functions* instead of *values*. There is no explicit representation of state, there are no assignment statments, and hence, no variables. Owing to the lack of state, FP functions are free from side-effects; so we say the FP is *applicative*.

All functions take one argument and they are evaluated using the single FP operation, *application* (the colon ':' is the apply operator). For example, we read $+ : \langle 3\ 4 \rangle$ as "apply the function '+' to its argument $\langle 3\ 4 \rangle$".

Functional programs express a functional-level combination of their components instead of describing state changes using value-oriented expressions. For example, we write the function returning the *sin* of the *cos* of its input, i.e., $sin(cos(x))$, as: sin @ cos. This is a *functional expression*, consisting of the single *combining form* called *compose* ('@' is the compose operator) and its *functional arguments sin* and *cos*.

All combining forms take functions as arguments and return functions as results; functions may either be applied, e.g., sin@ cos : 3, or used as a functional argument in another functional expression, e.g., *tan @ sin @ cos*.

As we have seen, FP's combining forms allow us to express control abstractions without the use of variables. The *apply to all* functional form (&) is another case in point. The function '& exp' exponentiates all the elements of its argument:

$$\&exp : \langle 1.0\ 2.0 \rangle \equiv \langle 2.718\ 7.389 \rangle \tag{1.1}$$

In (1.1) there are no induction variables, nor a loop bounds specification. Moreover, the code is useful for any size argument, so long as the sub-elements of its argument conform to the domain of the *exp* function.

We must change our view of the programming process to adapt to the functional style. Instead of writing down a set of steps that manipulate and assign values, we compose functional expressions using the higher-level functional forms. For example, the function that adds a scalar to all elements of a vector will be written in two steps. First, the function that distributes the scalar amongst each element of the vector:

$$distl : \langle 3\ \langle 4\ 6 \rangle \rangle \equiv \langle \langle 3\ 4 \rangle\ \langle 3\ 6 \rangle \rangle \tag{1.2}$$

Next, the function that adds the pairs of elements that make up a sequence:

$$\&+ : \langle \langle 3\ 4 \rangle\ \langle 3\ 6 \rangle \rangle \equiv \langle 7\ 9 \rangle \tag{1.3}$$

In a value-oriented programming language the computation would be expressed as:

$$\&+ : distl : \langle 3\ \langle 4\ 6 \rangle \rangle, \tag{1.4}$$

which means to apply 'distl' to the input and then to apply '+' to every element of the result. In FP we write (1.4) as:

$$\&+ \ @ \ distl : \langle 3\ \langle 4\ 6 \rangle \rangle. \tag{1.5}$$

The functional expression of (1.5) replaces the two step value expression of (1.4).

Often, functional expressions are built from the inside out, as in LISP. In the next example we derive a function that scales then shifts a vector, i.e., for scalars a, b and a vector $\overline{v}$, compute $a + b\overline{v}$. This FP function will have three arguments, namely a, b and $\overline{v}$. Of course, FP does not use formal parameter names, so they will be designated by the function symbols 1, 2, 3. The first code segment scales $\overline{v}$ by b (defintions are delimited with curly braces '{}'):

{scaleVec & @ distl @ [2,3]}* (1.6)

The code segment in (1.5) shifts the vector. The completed function is:

{changeVec &+ @ distl @ [1 , scaleVec]} (1.7)

In the derivation of the program we wrote from right to left, first doing the *distl*'s and then composing with the *apply-to-all* functional form. Using an imperative language, such as Pascal, we would write the program from the outside in, writing the loop before inserting the arithmetic operators.

Although FP encourages a recursive programming style, it provides combining forms to avoid explicit recursion. For example, the right insert combining form (!) can be used to write a function that adds up a list of numbers:

!+ : <1 2 3> ≡ 6 (1.8)

The equivalent, recursive function is much longer:

{addNumbers (null -> %0 ; + @ [1, addNumbers @ tl])} (1.9)

The generality of the combining forms encourages hierarchical program development. Unlike APL, which restricts the use of combining forms to certain builtin functions, FP allows combining forms to take any functional expression as an argument.

2. System Description

2.1. Objects

The set of objects Ω consists of the atoms and sequences $<x_1, x_2, \ldots, x_k>$ (where the $x_i \in \Omega$). (Lisp users should note the similarity to the list structure syntax, just replace the parenthesis by angle brackets and commas by blanks. There are no 'quoted' objects, i.e., 'abc). The atoms uniquely determine the set of valid objects and consist of the numbers (of the type found in FRANZ LISP [Fod80]), quoted ascii strings ("abcd"), and unquoted alphanumeric strings (abc3). There are three predefined atoms, **T** and **F**, that correspond to the logical values 'true' and 'false', and the undefined atom **?**, *bottom*. *Bottom* denotes the value returned as the result of an undefined operation, e.g., division by zero. The empty sequence, <> is also an atom. The following are examples of valid FP objects:

?	1.47	3888888888888
ab	"*CD*"	<1,<2,3>>
<>	**T**	<*a*,<>>

There is one restriction on object construction: no object may contain the undefined atom, such an object is itself undefined, e.g., $<1,?> \equiv ?$. This property is the so-called "bottom preserving property" [Ba78].

2.2. Application

This is the single FP operation and is designated by the colon (":"). For a function σ and an object x, $\sigma{:}x$ is an application and its meaning is the object that results from applying σ to x (i.e., evaluating $\sigma(x)$). We say that σ is the *operator* and that x is the *operand*. The following are examples of applications:

+:<7,8>	≡	15	tl:<1,2,3>	≡	<2,3>
1 :<*a*,*b*,*c*,*d*>	≡	*a*	2 :<*a*,*b*,*c*,*d*>	≡	*b*

2.3. Functions

All functions (F) map objects into objects, moreover, they are *strict*:

$$\sigma : ? \equiv ?, \ \forall \ \sigma \in \mathbf{F} \tag{2.1}$$

To formally characterize the primitive functions, we use a modification of McCarthy's conditional expressions [Mc60]:

$$p_1 \rightarrow e_1 ; \cdots ; p_n \rightarrow e_n ; e_{n+1} \tag{2.2}$$

This statement is interpreted as follows: return function e_1 if the predicate 'p_1' is true , . . . , e_n if 'p_n' is true. If none of the predicates are satisfied then default to e_{n+1}. It is assumed that x, x_i, y, y_i, $z_i \in \Omega$.

2.3.1. Structural

Selector Functions

For a nonzero integer μ,

$\mu : x \equiv$
$\quad x=\langle x_1, x_2, \ldots, x_k\rangle \wedge 0 < \mu \leq k \rightarrow x_\mu;$
$\quad x=\langle x_1, x_2, \ldots, x_k\rangle \wedge -k \leq \mu < 0 \rightarrow x_{k+\mu+1};$?

pick : $\langle n, x\rangle \equiv$
$\quad x=\langle x_1, x_2, \ldots, x_k\rangle \wedge 0 < n \leq k \rightarrow x_n;$
$\quad x=\langle x_1, x_2, \ldots, x_k\rangle \wedge -k \leq n < 0 \rightarrow x_{k+n+1};$?

The user should note that the function symbols **1**,**2**,**3**,... are to be distinguished from the atoms 1,2,3,....

last : $x \equiv$
$\quad x=\langle\rangle \rightarrow \langle\rangle ;$
$\quad x=\langle x_1, x_2, \ldots, x_k\rangle \wedge k \geq 1 \rightarrow x_k;$?

first : $x \equiv$
$\quad x=\langle\rangle \rightarrow \langle\rangle ;$
$\quad x=\langle x_1, x_2, \ldots, x_k\rangle \wedge k \geq 1 \rightarrow x_1;$?

Tail Functions

tl : $x \equiv$
$\quad x=\langle x_1\rangle \rightarrow \langle\rangle ;$
$\quad x=\langle x_1, x_2, \ldots, x_k\rangle \wedge k \geq 2 \rightarrow \langle x_2, \ldots, x_k\rangle ;$?

tlr : $x \equiv$
$\quad x=\langle x_1\rangle \rightarrow \langle\rangle ;$
$\quad x=\langle x_1, x_2, \ldots, x_k\rangle \wedge k \geq 2 \rightarrow \langle x_1, \ldots, x_{k-1}\rangle ;$?

Note: There is also a function **front** that is equivalent to **tlr**.

Distribute from left and right

distl : $x \equiv$
$\quad x=\langle y, \langle\rangle\rangle \rightarrow \langle\rangle;$
$\quad x=\langle y, \langle z_1, z_2, \ldots, z_k\rangle\rangle \rightarrow \langle\langle y, z_1\rangle, \ldots, \langle y, z_k\rangle\rangle;$?

distr : $x \equiv$

$x=<<>,y> \rightarrow <>;$

$x=<<y_1, y_2, \ldots, y_k>,z> \rightarrow <<y_1,z>,\ldots,<y_k,z>>;$?

Identity

id : $x \equiv x$

out : $x \equiv x$

Out is similar to id. Like id it returns its argument as the result, unlike id it prints its result on *stdout* – It is the only function with a side effect. *Out* is intended to be used for debugging only.

Append left and right

apndl : $x \equiv$

$x=<y,<>> \rightarrow <y>;$

$x=<y,<z_1, z_2, \ldots, z_k>> \rightarrow <y,z_1, z_2, \ldots, z_k>;$?

apndr : $x \equiv$

$x=<<>,z> \rightarrow <z>;$

$x=<<y_1, y_2, \ldots, y_k>,z> \rightarrow <y_1, y_2, \ldots, y_k, z>;$?

Transpose

trans : $x \equiv$

$x=<<>,\ldots,<>> \rightarrow <>;$

$x=<x_1, x_2, \ldots, x_k> \rightarrow <y_1, \ldots, y_m>;$?

where $x_i = <x_{i1}, \ldots, x_{im}> \wedge y_j = <x_{1j}, \ldots, x_{kj}>,$

$1 \leq i \leq k$, $1 \leq j \leq m.$

reverse : $x \equiv$

$x=<> \rightarrow;$

$x=<x_1, x_2, \ldots, x_k> \rightarrow <x_k, \ldots, x_1>;$?

Rotate Left and Right

rotl : $x \equiv$

$x=<> \rightarrow <>;\ x=<x_1> \rightarrow <x_1>;$

$x=<x_1, x_2, \ldots, x_k> \wedge k \geq 2 \rightarrow <x_2, \ldots, x_k, x_1>;$?

rotr : $x \equiv$

$x=<> \rightarrow <>;\ x=<x_1> \rightarrow <x_1>;$

$x=<x_1, x_2, \ldots, x_k> \wedge k \geq 2 \rightarrow <x_k, x_1, \ldots, x_{k-2}, x_{k-1}>;$?

concat : $x \equiv$

$x=<<x_{11}, \ldots, x_{1k}>,<x_{21}, \ldots, x_{2n}>,\ldots,<x_{m1}, \ldots, x_{mp}>> \wedge k, m, n, p > 0 \rightarrow$
$<x_{11}, \ldots, x_{1k}, x_{21}, \ldots, x_{2n}, \ldots, x_{m1}, \ldots, x_{mp}>; ?$

Concatenate removes all occurrences of the null sequence:

concat : <<1,3>,<>,<2,4>,<>,<5>> ≡ <1,3,2,4,5> (2.3)

pair : $x \equiv$

$x=<x_1, x_2, \ldots, x_k> \wedge k>0 \wedge k$ *is even* $\rightarrow <<x_1,x_2>, \ldots, <x_{k-1},x_k>>;$
$x=<x_1, x_2, \ldots, x_k> \wedge k>0 \wedge k$ *is odd* $\rightarrow <<x_1,x_2>, \ldots, <x_k>>; ?$

split : $x \equiv$

$x=<x_1> \rightarrow <<x_1>,<>>;$
$x=<x_1, x_2, \ldots, x_k> \wedge k>1 \rightarrow <<x_1, \ldots, x_{\lceil k/2 \rceil}>,<x_{\lceil k/2 \rceil+1}, \ldots, x_k>>; ?$

iota : $x \equiv$

$x=0 \rightarrow <>;$
$x \in N^+ \rightarrow <1,2,\ldots,x>; ?$

2.3.2. Predicate (Test) Functions

atom : $x \equiv x \in$ *atoms* $\rightarrow$ T; $x \neq ? \rightarrow$ F; ?

eq : $x \equiv x = <y,z> \wedge y=z \rightarrow$ T; $x=<y,z> \wedge y \neq z \rightarrow$ F; ?

Also less than (<), greater than (>), greater than or equal (>=), less than or equal (<=), not equal (~=); '=' is a synonym for eq.

null : $x \equiv x=<> \rightarrow$ T; $x \neq ? \rightarrow$ F; ?

length : $x \equiv x = <x_1, x_2, \ldots, x_k> \rightarrow k$; $x=<> \rightarrow 0$; ?

2.3.3. Arithmetic/Logical

+ : $x \equiv x=<y,z> \wedge y,z$ *are numbers* $\rightarrow y+z$; ?
− : $x \equiv x=<y,z> \wedge y,z$ *are numbers* $\rightarrow y-z$; ?
* : $x \equiv x=<y,z> \wedge y,z$ *are numbers* $\rightarrow y \times z$; ? / : $x \equiv x=<y,z> \wedge y,z$ *are numbers* $\wedge z \neq 0 \rightarrow y \div z$; ?

And, or, not, xor

and : $<x,y> \equiv x=$T $\rightarrow y$; $x=$F $\rightarrow$ F; ?

or : $<x,y> \equiv x=$F $\rightarrow y$; $x=$T $\rightarrow$ T; ?

xor : $<x,y> \equiv$
$x=\mathbf{T} \wedge y=\mathbf{T} \rightarrow \mathbf{F}; x=\mathbf{F} \wedge y=\mathbf{F} \rightarrow \mathbf{F};$
$x=\mathbf{T} \wedge y=\mathbf{F} \rightarrow \mathbf{T}; x=\mathbf{F} \wedge y=\mathbf{T} \rightarrow \mathbf{T}; ?$

not : $x \equiv x=\mathbf{T} \rightarrow F; x=\mathbf{F} \rightarrow \mathbf{T}; ?$

2.3.4. Library Routines

sin : $x \equiv x$ is a number $\rightarrow sin(x); ?$

asin : $x \equiv x$ is a number $\wedge\ |x| \leq 1 \rightarrow \sin^{-1}(x); ?$

cos : $x \equiv x$ is a number $\rightarrow cos(x); ?$

acos : $x \equiv x$ is a number $\wedge\ |x| \leq 1 \rightarrow \cos^{-1}(x); ?$

exp : $x \equiv x$ is a number $\rightarrow e^{x}; ?$

log : $x \equiv x$ is a positive number $\rightarrow ln(x); ?$

mod : $<x,y> \equiv x$ and y are numbers $\rightarrow x - y \times \left\lfloor \frac{x}{y} \right\rfloor; ?$

2.4. Functional Forms

Functional forms define new *functions* by operating on function and object *parameters* of the form. The resultant expressions can be compared and contrasted to the *value*-oriented expressions of traditional programming languages. The distinction lies in the domain of the operators; functional forms manipulate functions, while traditional operators manipulate values.

One functional form is *composition*. For two functions ϕ and ψ the form ϕ @ ψ denotes their composition $\phi \cdot \psi$:

$$(\phi \ @ \ \psi) : x \equiv \phi:(\psi:x), \ \forall \quad x \in \Omega \tag{2.4}$$

The *constant* function takes an object parameter:

$$\%x:y \equiv y=? \rightarrow ?; x, \ \forall \quad x,y \in \Omega \tag{2.5}$$

The function $\%?$ always returns ?.

In the following description of the functional forms, we assume that ξ, ξ_i, σ, σ_i, τ, and τ_i are functions and that x, x_i, y are objects.

Composition

$(\sigma \ @ \ \tau):x \equiv \sigma:(\tau:x)$

Construction

$[\sigma_1, \ldots, \sigma_n]:x \equiv <\sigma_1:x,\ldots,\sigma_n:x>$

Note that construction is also bottom-preserving, e.g.,

$$[+,/] : <3,0> = <3,?> = ? \tag{2.6}$$

Condition

$(\xi \text{ -> } \sigma; \tau):x \equiv$
$(\xi:x)=\mathbf{T} \rightarrow \sigma:x;$
$(\xi:x)=\mathbf{F} \rightarrow \tau:x; ?$

The reader should be aware of the distinction between *functional expressions*, in the variant of McCarthy's conditional expression, and the *functional form* introduced here. In the former case the result is a *value*, while in the latter case the result is a *function*. Unlike Backus' FP, the conditional form *must* be enclosed in parenthesis, e.g.,

$$\text{(isNegative -> - @ [\%0,id] ; id)} \tag{2.7}$$

Constant

$\%x:y \equiv y=? \rightarrow ?; x, \quad \forall\ x \in \Omega$

This function returns its object parameter as its result.

Right Insert

$!\sigma :x \equiv$
$x=<> \rightarrow e_f :x;$
$x=<x_1> \rightarrow x_1;$
$x=<x_1, x_2, \ldots, x_k> \wedge k>2 \rightarrow \sigma:<x_1, !\sigma:<x_2, \ldots, x_k>>; ?$

e.g., $!+:<4,5,6>=15$.

If σ has a right identity element e_f, then $!\sigma:<> = e_f$, e.g.,

$$!+ : <>=0 \text{ and } !* : <>=1 \tag{2.8}$$

Currently, identity functions are defined for + (0), − (0), * (1), / (1), also for **and** (T), **or** (F), **xor** (F). All other unit functions default to bottom (?).

Tree Insert

$$
\begin{aligned}
&|\sigma : x \equiv \\
&\quad x=<> \rightarrow e_f : x; \\
&\quad x=<x_1> \rightarrow x_1; \\
&\quad x=<x_1, x_2, \ldots, x_k> \wedge k>1 \rightarrow \\
&\quad \sigma : <|\sigma : <x_1, \ldots, x_{\lceil k/2 \rceil}> , |\sigma : <x_{\lceil k/2 \rceil+1}, \ldots, x_k>>; ?
\end{aligned}
$$

e.g.,

$$|+:<4,5,6,7> \equiv +:<+:<4,5>,+:<6,7>> \equiv 15 \qquad (2.9)$$

Tree insert uses the same identity functions as right insert.

Apply to All

$$
\begin{aligned}
&\&\sigma: x \equiv \\
&\quad x=<> \rightarrow <>; \\
&\quad x=<x_1, x_2, \ldots, x_k> \rightarrow <\sigma:x_1, \ldots, \sigma:x_k>; ?
\end{aligned}
$$

While

$$
\begin{aligned}
&(\text{while } \xi\ \sigma):x \equiv \\
&\quad \xi:x=\mathbf{T} \rightarrow (\text{while } \xi\ \sigma):(\sigma:x); \\
&\quad \xi:x=\mathbf{F} \rightarrow x; ?
\end{aligned}
$$

2.5. User Defined Functions

An FP definition is entered as follows:

$$\{\textit{fn-name fn-form}\}, \qquad (2.10)$$

where *fn-name* is an ascii string consisting of letters, numbers and the underline symbol, and *fn-form* is any valid functional form, including a single primitive or defined function. For example the function

$$\{\textit{factorial } !* @ \textit{iota}\} \qquad (2.11)$$

is the non-recursive definition of the factorial function. Since FP systems are applicative it is permissible to substitute the actual definition of a function for any reference to it in a functional form: if $f \equiv 1@2$ then $f : x \equiv 1@2 : x, \ \forall\ x \in \Omega$.

References to undefined functions bottom out:

$$f:x \equiv ? \forall\ x \in \Omega,\ f \notin \mathbf{F} \qquad (2.12)$$

3. Getting on and off the System

Startup FP from the shell by entering the command:

/usr/local/fp.

The system will prompt you for input by indenting over six character positions. Exit from FP (back to the shell) with a control/D (^D).

3.1. Comments

A user may end any line (including a command) with a comment; the comment character is '#'. The interpreter will ignore any character after the '#' until it encounters a newline character or end-of-file, whichever comes first.

3.2. Breaks

Breaks interrupt any work in progress causing the system to do a FRANZ reset before returning control back to the user.

3.3. Non-Termination

LISP's namestack may, on occasion, overflow. FP responds by printing "non-terminating" and returning bottom as the result of the application. It does a FRANZ reset before returning control to the user.

4. System Commands

System commands start with a right parenthesis and they are followed by the command-name and possibly one or more arguments. All this information *must be typed on a single line*, and any number of spaces or tabs may be used to separate the components.

4.1. Load

Redirect the standard input to the file named by the command's argument. If the file doesn't exist then FP appends '.fp' to the file-name and retries the open (error if the file doesn't exist). This command allows the user to read in FP function definitions from a file. The user can also read in applications, but such operation is of little utility since none of the input is echoed at the terminal. Normally, FP returns control to the user on an end-of-file. It will also do so whenever it does a FRANZ reset, e.g., whenever the user issues a break, or whenever the system encounters a non-terminating application.

4.2. Save

Output the source text for all user-defined functions to the file named by the argument.

4.3. Csave and Fsave

These commands output the lisp code for all the user-defined functions, including the original source-code, to the file named by the argument. Csave pretty prints the code, Fsave does not. Unless the user wishes to examine the code, he should use 'fsave'; it is about ten times faster than 'csave', and the resulting file will be about three times smaller.

These commands are intended to be used with the liszt compiler and the 'cload' command, as explained below.

4.4. Cload

This command loads or fasls in the file shown by the argument. First, FP appends a '.o' to the file-name, and attempts a load. Failing that, it tries to load the file named by the argument. If the user outputs his function definitions using fsave or csave, and then compiles them using liszt, then he may fasl in the compiled code and speed up the execution of his defined functions by a factor of 5 to 10.

4.5. Pfn

Print the source text(s) (at the terminal) for the user-defined function(s) named by the argument(s) (error if the function doesn't exist).

4.6. Delete

Delete the user-defined function(s) named by the argument (error if the function doesn't exist).

4.7. Fns

List the names of all user-defined functions in alphabetical order. Traced functions are labeled by a trailing '@' (see § 4.7 for sample output).

4.8. Stats

The "stats" command has several options that help the user manage the collection of dynamic statistics for functions[1] and functional forms. Option names follow the keyword "stats", e.g., ")stats reset".

The statistic package records the frequency of usage for each function and functional form; also the size[2] of all the arguments for all functions and functional expressions. These two measures allow the user to derive the average argument size per call. For functional forms the package tallies the frequency of each functional argument. Construction has an additional statistic that tells the number of functional arguments involved in the construction.

Statistics are gathered whenever the mode is on, except for applications that "bottom out" (i.e., return bottom – ?). Statistic collection slows the system down by ×2 to ×4. The following printout illustrates the use of the statistic package (user input is emboldened):

[1] Measurement of user-defined functions is done with the aid of the trace package, discussed in § 4.9.

[2] "Size" is the top-level length of the argument, for most functions. Exceptions are: *apndl, distl* (top-level length of the second element), *apndr, distr* (top-level length of the first element), and *transpose* (top level length of each top level element).

```
)stats on

Stats collection turned on.

+:<3 4>
7
!* @ iota :3
6
)stats print

plus:       times    1

times:      times    2

iota:       times    1

insert:     times    1              size  3

                  Functional Args
            Name                              Times
            times                   1

compos:     times    1              size  1

                  Functional Args
            Name                              Times
            insert                  1
            iota                              1
```

4.8.1. On

Enable statistics collection.

4.8.2. Off

Disable statistics collection. The user may selectively collect statistics using the on and off commands.

4.8.3. Print

Print the dynamic statistics at the terminal, or, output them to a file. The latter option requires an additional argument, e.g., ")stats print fooBar" prints the stats to the file "fooBar".

4.8.4. Reset

Reset the dynamic statistics counters. To prevent accidental loss of collected statistics, the system will query the user if he tries to reset the counters without first outputting the data (the system will also query the user if he tries to log out without outputting the data).

4.9. Trace

Enable or disable the tracing and the dynamic measurement of the user defined functions named by the argument(s). The first argument tells whether to turn tracing off or on and the others give the name of the functions affected. The tracing and untracing commands are independent of the dynamic statistics commands. This command is cumulative e.g., ')trace on f1', followed by ')trace on f2' is equivalent to ')trace on f1 f2'.

FP tracer output is similar to the FRANZ tracer output: function entries and exits, call level, the functional argument (remember that FP functions have only one argument!), and the result, are printed at the terminal:

```
                )pfn fact

{fact (eq0 -> %1 ; * @ [id, fact @ s1])}
                )fns

eq0             fact  s1

                )trace on fact
                )fns

eq0             fact@     s1

                fact : 2

1 >Enter> fact [2]
|2 >Enter> fact [1]
||3 >Enter> fact [0]
||3 <EXIT< fact 1
|2 <EXIT< fact 1
1 <EXIT< fact 2

2
```

4.10. Timer

FP provides a simple timing facility to time top-level applications. The command ")timer on" puts the system in timing mode, ")timer off" turns the mode off (the mode is initially off). While in timing mode, the system reports CPU time, garbage collection time, and elapsed time, in seconds. The timing output follows the printout of the result of the application.

4.11. Script

Open or close a script file. The first argument gives the option, the second the optional script file-name. The "open" option causes a new script-file to be opened and any currently open script file to be closed. If the file cannot be opened, FP sends and error message and, if a script file was already opened, it remains open. The command ")script close" closes an open script file. The user may elect to append script output to the script-file with the append mode.

4.12. Help

Print a short summary of all the system commands:

```
    )help
Commands are:

load <file>                     Redirect input from <file>
save <file>                     Save defined fns in <file>
pfn <fn1> ...                   Print source text of <fn1> ...
delete <fn1> ...                Delete <fn1> ...
fns                             List all functions
stats on/off/reset/print [file] Collect and print dynamic stats
trace on/off <fn1> ...          Start/Stop exec trace of <fn1> ...
timer on/of                     Turn timer on/off
script open/close/append        Open or close a script-file
lisp                            Exit to the lisp system (return with '^D')
debug on/off                    Turn debugger output on/off
csave <file>                    Output Lisp code for all user-defined fns
cload <file>                    Load Lisp code from a file (may be compiled)
fsave <file>                    Same as csave except without pretty-printing
```

4.13. Special System Functions

There are two system functions that are not generally meant to be used by average users.

4.13.1. Lisp

This exits to the lisp system. Use "^D" to return to FP.

4.13.2. Debug

Turns the 'debug' flag on or off. The command ")debug on" turns the flag on, ")debug off" turns the flag off. The main purpose of the command is to print out the parse tree.

5. Programming Examples

We will start off by developing a larger FP program, *mergeSort*. We measure *mergeSort* using the trace package, and then we comment on the measurements. Following *mergeSort* we show an actual session at the terminal.

5.1. MergeSort

The source code for *mergeSort* is:

```
# Use a divide and conquer strategy
{mergeSort | merge}

{merge atEnd @ mergeHelp @ [[], fixLists]}

# Must convert atomic arguments into sequences
# Atomic arguments occur at the leaves of the execution tree
{fixLists &(atom -> [id] ; id)}

# Merge until one or both input lists are empty
{mergeHelp (while and @ &(not@null) @ 2
                  (firstIsSmaller -> takeFirst ;
                                     takeSecond))}

# Find the list with the smaller first element
{firstIsSmaller < @ [1@1@2, 1@2@2]}

# Take the first element of the first list
{takeFirst [apndr@[1,1@1@2], [tl@1@2, 2@2]]}

# Take the first element of the second list
{takeSecond [apndr@[1,1@2@2], [1@2, tl@2@2]]}

# If one list isn't null, then append it to the
# end of the merged list
{atEnd (firstIsNull -> concat@[1,2@2] ;
                       concat@[1,1@2])}

{firstIsNull null@1@2}
```

The merge sort algorithm uses a divide and conquer strategy; it splits the input in half, recursively sorts each half, and then merges the sorted lists. Of course, all these sub-sorts can execute in parallel, and the tree-insert ([]) functional form expresses this concurrency. *Merge* removes successively larger elements from the heads of the two lists (either *takeFirst* or *takeSecond*) and appends these elements to the end of the merged sequence. *Merge* terminates when one sequence is empty, and then *atEnd* appends any remaining non-empty sequence to the end of the merged one.

On the next page we give the trace of the function *merge*, which information we can use to determine the structure of *merge*'s execution tree. Since the tree is well-balanced, many of the *merge*'s could be executed in parallel. Using this trace we can also calculate the average length of the arguments passed to *merge*, or a distribution of argument lengths. This information is useful for determining communication costs.

```
)trace on merge

      mergeSort : <0 3 -2 1 11 8 -22 -33>
|3 >Enter> merge [<0 3>]
|3 <EXIT<  merge  <0 3>
|3 >Enter> merge [<-2 1>]
|3 <EXIT<  merge  <-2 1>
|2 >Enter> merge [<<0 3> <-2 1>>]
|2 <EXIT<  merge  <-2 0 1 3>
|3 >Enter> merge [<11 8>]
|3 <EXIT<  merge  <8 11>
|3 >Enter> merge [<-22 -33>]
|3 <EXIT<  merge  <-33 -22>
|2 >Enter> merge [<<8 11> <-33 -22>>]
|2 <EXIT<  merge  <-33 -22 8 11>
1 >Enter> merge [<<-2 0 1 3> <-33 -22 8 11>>]
1 <EXIT<  merge  <-33 -22 -2 0 1 3 8 11>

<-33 -22 -2 0 1 3 8 11>
```

5.2. FP Session

User input is **emboldened**, terminal output in Roman script.

fp

```
FP, v. 4.1 11/31/82
        )load ex_man
{all_le}
{sort}
{abs_val}
{find}
{ip}
{mm}
{eq0}
{fact}
{sub1}
{alt_fnd}
{alt_fact}
        )fns

abs_val   all_le   alt_fact   alt_fnd   eq0   fact   find
ip        mm       sort       sub1
```

abs_val : 3

```
3
```

abs_val : -3

```
3
```

abs_val : 0

```
0
```

abs_val : <-5 0 66>

```
?
```

&abs_val : <-5 0 66>

```
<5 0 66>
```

)pfn abs_val

```
{abs_val ((> @ [id,%0]) -> id ; (- @ [%0,id]))}
```

[id,%0] : -3

```
<-3 0>
```

[%0,id] : -3

```
<0 -3>
        - @ [%0,id] : -3
3
        all_le : <1 3 5 7>
T
        all_le : <1 0 5 7>
F
        )pfn all_le
{all_le ! and @ &<= @ distl @ [1,tl]}
        distl @ [1,tl] : <1 2 3 4>
<<1 2> <1 3> <1 4>>
        &<= @ distl @ [1,tl] : <1 2 3 4>
<T T T>
        ! and : <F T T>
F
        ! and : <T T T>
T
        sort : <3 1 2 4>
<1 2 3 4>
        sort : <1>
<1>
        sort : <>
?
        sort : 4
?
        )pfn sort
{sort (null @ tl -> [1] ; (all_le -> apndl @ [1,sort@tl]; sort@rotl))}
        fact : 3
6
```

```
        )pfn fact sub1 eq0
{fact (eq0 -> %1 ; *@[id , fact@sub1])}
{sub1 -@[id,%1]}
{eq0 = @ [id,%0]}
        &fact : <1 2 3 4 5>
<1 2 6 24 120>
        eq0 : 3
F
        eq0 : <>
F
        eq0 : 0
T
        sub1 : 3
2
        %1 : 3
1
        alt_fact : 3
6
        )pfn alt_fact
{alt_fact !* @ iota}
        iota : 3
<1 2 3>
        !* @ iota : 3
6
        !+ : <1 2 3>
6
        find : <3 <3 4 5>>
T
        find : <<> <3 4 <>>>
```

```
T
        find : <3 <4 5>>
F
        )pfn find
{find (null@2 -> %F ; (=@[1,1@2] -> %T ; find@[1,tl@2]))}
        [1,tl@2] : <3 <3 4 5>>
<3 <4 5>>
        [1,1@2] : <3 <3 4 5>>
<3 3>
        alt_fnd : <3 <3 4 5>>
T
        )pfn alt_fnd
{alt_fnd ! or @ &eq @ distl }
        distl : <3 <3 4 5>>
<<3 3> <3 4> <3 5>>
        &eq @ distl : <3 <3 4 5>>
<T F F>
        !or : <T F T>
T
        !or : <F F F>
F
        )delete alt_fnd
        )fns
abs_val   all_le   alt_fact   eq0   fact   find   ip
mm        sort     subl
        alt_fnd : <3 <3 4 5>>
alt_fnd not defined
?
        {g g}
```

```
{g}
        g : 3

non-terminating

?

[Return to top level]

FP, v. 4.0 10/8/82
        [+,*] : <3 4>

<7 12>

            [+,* : <3 4>

    syntax error:

    [+,* : <3 4>
           ^

        ip : <<3 4 5> <5 6 7>>

74

        )pfn ip

{ip !+ @ &* @ trans}

        trans : <<3 4 5> <5 6 7>>

<<3 5> <4 6> <5 7>>

        &* @ trans : <<3 4 5> <5 6 7>>

<15 24 35>

        mm : <<<1 0> <0 1>> <<3 4> <5 6>>>

<<3 4> <5 6>>

        )pfn mm

{mm &&ip @ &distl @ distr @[1,trans@2]}

        [1,trans@2] : <<<1 0> <0 1>> <<3 4> <5 6>>>

<<<1 0> <0 1>> <<3 4> <5 6>>>

        distr : <<<1 0> <0 1>> <<3 4> <5 6>>>

<<<1 0> <<3 4> <5 6>>> <<0 1> <<3 4> <5 6>>>>

        &distl : <<<1 0> <<3 4> <5 6>>> <<0 1> <<3 4> <5 6>>>>

<<<<1 0> <3 4>> <<1 0> <5 6>>> <<<0 1> <3 4>> <<0 1> <5 6>>>>
```

```
        &ip @ &dist & distr @ [1,trans @ 2] : <<<1 0> <0 1>> <<3 4> <5 6>>>
    syntax error:
    [+,* : <3 4>
    &ip @ &distl & distr @ [1,trans @ 2] : <<<1 0> <0 1>> <<3 4> <5 6>>>
        &ip @ &distl @ distr @ [1,trans@2] : <<<1 0> <0 1>> <<3 4> <5 6>>>
?
```

6. Implementation Notes

FP was written in 3000 lines of FRANZ LISP [Fod 80]. Table 1 breaks down the distribution of the code by functionality.

Functionality	% (bytes)
compiler	34
user interface	32
dynamic stats	16
primitives	14
miscellaneous	3

Table 1

6.1. The Top Level

The top-level function *runFp* starts up the subsystem by calling the routine *fpMain*, that takes three arguments:

(1) A boolean argument that says whether debugging output will be enabled.

(2) A Font identifier. Currently the only one is supported **'asc** (ASCII).

(3) A boolean argument that identifies whether the interpreter was invoked from the shell. If so then all exits from FP return the user back to the shell.

The compiler converts the FP functions into LISP equivalents in two stages: first it forms the parse tree, and then it does the code generation.

6.2. The Scanner

The scanner consists of a main routine, *get_tkn*, and a set of action functions. There exists one set of action functions for each character font (currently only ASCII is supported). All the action functions are named *scan*$<*font*>, where <*font*> is the specified font, and each is keyed on a particular character (or sometimes a particular character-type – e.g., a letter or a number). *get_tkn* returns the token type, and any ancillary information, e.g., for the token "name" the name itself will also be provided. (See Appendix C for the font-token name correspondences). When a character has been read the scanner finds the action function by doing a

(*get* '*scan*$ <*font*> <*char*>)

A syntax error message will be generated if no action exists for the particular character read.

6.3. The Parser

The main parsing function, *parse*, accepts a single argument, that identifies the parsing context, or type of construct being handled. Table 2 shows the valid parsing contexts.

id	construct
top_lev	initial call
constr$$	construction
compos$$	composition
alpha$$	apply-to-all
insert$$	insert
ti$$	tree insert
arrow$$	affirmative clause of conditional
semi$$	negative clause of conditional
lparen$$	parenthetic expr.
while$$	while

Table 2, Valid Parsing Contexts

For each type of token there exists a set of parse action functions, of the name *p$<tkn-name>*. Each parse-action function is keyed on a valid context, and it is looked up in the same manner as scan action functions are looked up. If an action function cannot be found, then there is a syntax error in the source code. Parsing proceeds as follows: initially *parse* is called from the top-level, with the context argument set to *"top_lev"*. Certain tokens cause parse to be recursively invoked using that token as a context. The result is the parse tree.

6.4. The Code Generator

The system compiles FP source into LISP source. Normally, this code is interpreted by the FRANZ LISP system. To speed up the implementation, there is an option to compile into machine code using the *liszt* compiler [Joy 79]. This feature improves performance tenfold, for some programs.

The compiler expands all functional forms into their LISP equivalents instead of inserting calls to functions that generate the code at run-time. Otherwise, *liszt* would be ineffective in speeding up execution since all the functional forms would be executed interpretively. Although the amount of code generated by an expanding compiler is 3 or 4 times greater than would be generated by a non-expanding compiler, even in interpreted mode the code runs twice as quickly as unexpanded code. With *liszt* compilation this performance advantage increases to more than tenfold.

A parse tree is either an atom or a hunk of parse trees. An atomic parse-tree identifies either an fp built-in function or a user defined function. The hunk-type parse tree represents functional forms, e.g., compose or construct. The first element identifies the functional form and the other elements are its functional parameters (they may in turn be functional forms). Table 3 shows the parse-tree formats.

Form	Format
user-defined	<atom>
fp builtin	<atom>
apply-to-all	{*alpha*$$ Φ}
insert	{*insert*$$ Φ}
tree insert	{*ti*$$ Φ}
select	{*select*$$ μ}
constant	{*constant*$$ μ}
conditional	{*condit*$$ Φ_1 Φ_2 Φ_3}
while	{*while*$$ Φ_1 Φ_2}
compose	{*compos*$$ Φ_1 Φ_2}
construct	{*constr*$$ Φ_1 Φ_2 , . . . , Φ_n *nil*}

Note: Φ and the Φ_k are parse-trees and μ is an optionally signed integer constant.

Table 3, Parse-Tree Formats

6.5. Function Definition and Application

Once the code has been generated, then the system defines the function via *putd*. The source code is placed onto a property list, *' sources*, to permit later access by the system commands.

For an application, the indicated function is compiled and then defined, only temporarily, as *tmp*$$. The single argument is read and *tmp*$$ is applied to it.

6.6. Function Naming Conventions

When the parser detects a named primitive function, it returns the name <*name*>*$fp*, where <*name*> is the name that was parsed (all primitive function-names end in *$fp*). See Appendix D for the symbolic (e.g., compose, +) function names.

Any name that isn't found in the list of builtin functions is assumed to represent a user-defined function; hence, it isn't possible to redefine FP primitive functions. FP protects itself from accidental or malicious internal destruction by appending the suffix "*_fp*" to all user-defined function names, before they are defined.

6.7. Measurement Impelementation

This work was done by Dorab Patel at UCLA. Most of the measurement code is in the file 'fpMeasures.l'. Many of the remaining changes were effected in 'primFp.l', to add calls on the measurement package at run-time; to 'codeGen.l', to add tracing of user defined functions; to 'utils.l', to add the new system commands; and to 'fpMain.l', to protect the user from forgetting to output statistics when he leaves FP.

6.7.1. Data Structures

All the statistics are in the property list of the global symbol *Measures*. Associated with each each function (primitive or user-defined, or functional form) is an indicator; the statistics gathered for each function are the corresponding values. The names corresponding to primitive functions and functional forms end in '$fp' and the names corresponding to user-defined functions end in '_fp'. Each of the property values is an association list:

(get 'Measures 'rotl$fp) ==> ((times . 0) (size . 0))

The car of the pair is the name of the statistic (i.e., times, size) and the cdr is the value. There is one exception. Functional forms have a statistic called funargtyp. Instead of being a dotted pair, it is a list of two elements:

```
(get 'Measures 'compose$fp) ==>
((times . 2) (size . 4) (funargtyp ((select$fp . 2) (sub$fp . 2))))
```

The car is the atom 'funargtyp' and the cdr is an alist. Each element of the alist consists of a functional argument-frequency dotted pair.

The statistic packages uses two other global symbols. The symbol DynTraceFlg is non-nil if dynamic statistics are being collected and is nil otherwise. The symbol TracedFns is a list (initially nil) of the names of the user functions being traced.

6.7.2. Interpretation of Data Structures

6.7.2.1. Times

The number of times this function has been called. All functions and functional forms have this statistic.

6.7.2.2. Size

The sum of the sizes of the arguments passed to this function. This could be divided by the times statistic to give the average size of argument this function was passed. With few exceptions, the size of an object is its top-level length (note: version 4.0 defined the size as the total number of *atoms* in the object); the empty sequence, "<>", has a size of 0 and all other atoms have size of one. The exceptions are: *apndl, distl* (top-level length of the second element), *apndr, distr* (top-level length of the first element), and *transpose* (top level length of each top level element).

This statistic is not collected for some primitive functions (mainly binary operators like +,-,*).

6.7.2.3. Funargno

The number of functional arguments supplied to a functional form.

Currently this statistic is gatherered only for the construction functional form.

6.7.2.4. Funargtyp

How many times the named function was used as a functional parameter to the particular functional form.

6.8. Trace Information

The level number of a call shows the number of steps required to execute the function on an ideal machine (i.e., one with unbounded resources). The level number is calculated under an assumption of infinite resources, and the system evaluates the condition of a conditional before evaluating either of its clauses. The number of functions executed at each level can give an idea of the distribution of parallelism in the given FP program.

7. Acknowledgements

Steve Muchnick proposed the initial construction of this system. Bob Ballance added some of is own insights, and John Foderaro provided helpful hints regarding effective use of the FRANZ LISP system [Fod80]. Dorab Patel [Pat81] wrote the dynamic trace and statistics package and made general improvements to the user interface. Portions of this manual were excerpted from the

COMPCON-83 Digest of Papers[3].

8. References

[Bac78]
John Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *CACM*, Turing Award Lecture, 21, 8 (August 1978), 613-641.

[Fod80]
John K. Foderaro, "The FRANZ LISP Manual," University of California, Berkeley, California, 1980.

[Joy79]
W.N. Joy, O. Babaoglu, "UNIX Programmer's Manual," November 7, 1979, Computer Science Division, University of California, Berkeley, California.

[Mc60]
J. McCarthy, "Recursive Functions of Symbolic expressions and their Computation by Machine," Part I, *CACM* 3, 4 (April 1960), 184-195.

[Pat80]
Dorab Ratan Patel, "A System Organization for Applicative Programming," M.S Thesis, University of California, Los Angeles, California, 1980.

[Pat81]
Dorab Patel, "Functional Language Interpreter User Manual," University of California, Los Angeles, California, 1981.

[3] Scott B. Baden and Dorab R. Patel, "Berkeley FP - Experiences With a Functional Programming Language", © 1982, IEEE.

Appendix A: Local Modifications

1. Character Set Changes

Backus [Ba78] used some characters that do not appear on our ASCII terminals, so we have made the following substitutions:

constant	$\bar{x}$	%x
insert	/	!
apply-to-all	α	&
composition	$\cdot$	@
arrow	$\rightarrow$	->
empty set	ϕ	<>
bottom	$\perp$	?
divide	$\div$	/
multiply	$\times$	*

2. Syntactic Modifications

2.1. While and Conditional

While and conditional functional expressions *must* be enclosed in parenthesis, e.g.,

(**while** *f g*)

(*p* -> *f* ; *g*)

2.2. Function Definitions

Function definitions are enclosed by curly braces; they consist of a name-definition pair, separated by blanks. For example:

{**fact** !* @ iota}

defines the function **fact** (the reader should recognize this as the non-recursive factorial function).

2.3. Sequence Construction

It is not necessary to separate elements of a sequences with a comma; a blank will suffice:

<1,2,3> ≡ <1 2 3>

For nested sequences, the terminating right angle bracket acts as the delimiter:

<<1,2,3>,<4,5,6>> ≡ <<1 2 3><4 5 6>>

3. User Interface

We have provided a rich set of commands that allow the user to catalog, print, and delete functions, to load them from a file and to save them away. The user may generate script files, dynamically trace and measure functional expression execution, generate debugging output, and, temporarily exit to the FRANZ LISP system. A command must begin with a right parenthesis. Consult Appendix C for a complete description of the command syntax.

Debugging in FP is difficult; all undefined results map to a single atom – *bottom* ("?"). To pinpoint the cause of an error the user can use the special debugging output function, **out**, or the tracer.

4. Additions and Ommissions

Many relational functions have been added: $<$, $>$, $=$, $\neq$, $\leq$, $\geq$; their syntax is: $<$, $>$, $=$, $\tilde{}=$, $<=$, $>=$. Also added are the **iota** function (This is the APL iota function an n-element sequence of natural numbers) and the exclusive OR ($\oplus$) function.

Several new structural functions have been added: **pair** pairs up successive elements of a sequence, **split** splits a sequence into two (roughly) equal halves, **last** returns the last element of the sequence ($<>$ if the sequence is empty), **first** returns the first element of the sequence ($<>$ if it is empty), and **concat** concatenates all subsequences of a sequence, squeezing out null sequences ($<>$). **Front** is equivalent to **tlr**. **Pick** is a parameterized form of the selector function; the first component of the argument selects a single element from the second component. **Out** is the only side-effect function; it is equivalent to the **id** function but it also prints its argument out at the terminal. This function is intended to be used only for debugging.

One new functional form has been added, tree insert. This functional form breaks up the the argument into two roughly equal pieces applying itself recursively to the two halves. The functional parameter is applied to the result.

The binary-to-unary functions ('**bu**') has been omitted.

Seven mathematical library functions have been added: sin, cos, asin ($\sin^{-1}$), acos ($\cos^{-1}$), log, exp, and mod (the remainder function)

Appendix B: FP Grammar

I. BNF Syntax

fpInput →	(fnDef | application | fpCmd[a])* | '^D'
fnDef →	'{' name funForm '}'
application →	funForm ':' object
name →	letter (letter | digit | '_')*
nameList →	(name)*
object →	atom | fpSequence | '?'
fpSequence →	'<' (ϵ | object (',' | ' ') object)*) '>'
atom →	'T' | 'F' | '<>' | '"' (ascii-char)* '"' | (letter | digit)* | number
funForm →	simpFn | composition | construction | conditional | constantFn | insertion | alpha | while | '(' funForm ')'
simpFn →	fpDefined | fpBuiltin
fpDefined →	name
fpBuiltin →	selectFn | 'tl' | 'id' | 'atom' | 'not' | 'eq' | relFn | 'null' | 'reverse' | 'distl' | 'distr' | 'length' | binaryFn | 'trans' | 'apndl' | 'apndr' | 'tlr' | 'rotl' | 'rotr' | 'iota' | 'pair' | 'split' | 'concat' | 'last' | 'libFn'
selectFn →	(ϵ | '+' | '-') unsignedInteger
relFn →	'<=' | '<' | '=' | '~=' | '>' | '>='
binaryFn →	'+' | '-' | '*' | '/' | 'or' | 'and' | 'xor'
libFn →	'sin' | 'cos' | 'asin' | 'acos' | 'log' | 'exp' | 'mod'
composition →	funForm '@' funForm
construction →	'[' formList ']'
formList →	ϵ | funForm (',' funForm)*
conditional →	'(' funForm '->' funForm ';' funForm ')'
constantFn →	'%' object
insertion →	'!' funForm | '|' funForm
alpha →	'&' funForm
while →	'(' 'while' funForm funForm ')'

II. Precedences

1.	%, !, &	(highest)
2.	@	
3.	[· · ·]	
4.	-> · · · ; · · ·	
5.	while	(least)

[a] Command Syntax is listed in Appendix C.

Appendix C: Command Syntax

All commands begin with a right parenthesis (")").

)fns
)pfn <nameList>
)load <UNIX file name>
)cload <UNIX file name>
)save <UNIX file name>
)csave <UNIX file name>
)fsave <UNIX file name>
)delete <nameList>
)stats on
)stats off
)stats reset
)stats print [UNIX file name]
)trace on <nameList>
)trace off <nameList>
)timer on
)timer off
)debug on
)debug off
)script open <UNIX file name>
)script close
)script append <UNIX file name>
)help
)lisp

Appendix D: Token-Name Correspondences

Token	Name
[	lbrack$$
]	rbrack$$
{	lbrace$$
}	rbrace$$
(	lparen$$
)	rparen$$
@	compos$$
!	insert$$
|	ti$$
&	alpha$$
;	semi$$
:	colon$$
,	comma$$
+	builtin$$
+ μ[a]	select$$
*	builtin$$
/	builtin$$
=	builtin$$
-	builtin$$
->	arrow$$
- μ	select$$
>	builtin$$
>=	builtin$$
<	builtin$$
<=	builtin$$
~=	builtin$$
%o[b]	constant$$

[a] μ is an optionally signed integer constant.

[b] o is any FP object.

Appendix E: Symbolic Primitive Function Names

The scanner assigns names to the alphabetic primitive functions by appending the string "$fp" to the end of the function name. The following table designates the naming assignments to the non-alphabetic primitive function names.

Function	Name
+	plus$fp
-	minus$fp
*	times$fp
/	div$fp
=	eq$fp
>	gt$fp
>=	ge$fp
<	lt$fp
<=	le$fp
~=	ne$fp

Yacc: Yet Another Compiler-Compiler

Stephen C. Johnson

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 - 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

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.

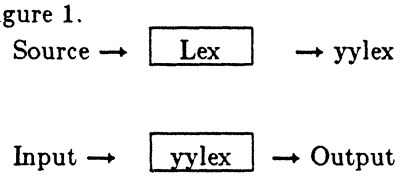
1. Introduction.

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.

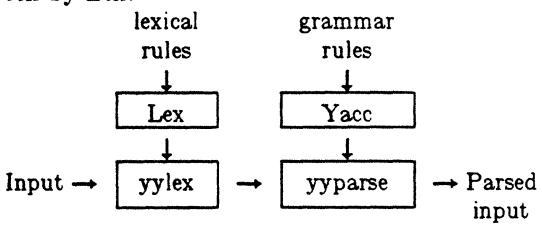
```
%%
[ \t]+$    ;
```

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.

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 – character 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 nonportable:

[\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 begin-

ning 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.

RATFOR — A Preprocessor for a Rational Fortran

Brian W. Kernighan

structured programming, control flow, programming

ABSTRACT

Although Fortran is not a pleasant language to use, it does have the advantages of universality and (usually) relative efficiency. The Ratfor language attempts to conceal the main deficiencies of Fortran while retaining its desirable qualities, by providing decent control flow statements:

- statement grouping
- **if-else** and **switch** for decision-making
- **while**, **for**, **do**, and **repeat-until** for looping
- **break** and **next** for controlling loop exits

and some "syntactic sugar":

- free form input (multiple statements/line, automatic continuation)
- unobtrusive comment convention
- translation of >, >=, etc., into .GT., .GE., etc.
- **return**(expression) statement for functions
- **define** statement for symbolic parameters
- **include** statement for including source files

Ratfor is implemented as a preprocessor which translates this language into Fortran.

Once the control flow and cosmetic deficiencies of Fortran are hidden, the resulting language is remarkably pleasant to use. Ratfor programs are markedly easier to write, and to read, and thus easier to debug, maintain and modify than their Fortran equivalents.

It is readily possible to write Ratfor programs which are portable to other env ironments. Ratfor is written in itself in this way, so it is also portable; versions of Ratfor are now running on at least two dozen different types of computers at over five hundred locations.

This paper discusses design criteria for a Fortran preprocessor, the Ratfor language and its implementation, and user experience.

September 16, 1986

RATFOR — A Preprocessor for a Rational Fortran

Brian W. Kernighan

structured programming, control flow, programming

1. INTRODUCTION

Most programmers will agree that Fortran is an unpleasant language to program in, yet there are many occasions when they are forced to use it. For example, Fortran is often the only language thoroughly supported on the local computer. Indeed, it is the closest thing to a universal programming language currently available: with care it is possible to write large, truly portable Fortran programs[1]. Finally, Fortran is often the most "efficient" language available, particularly for programs requiring much computation.

But Fortran *is* unpleasant. Perhaps the worst deficiency is in the control flow statements — conditional branches and loops — which express the logic of the program. The conditional statements in Fortran are primitive. The Arithmetic IF forces the user into at least two statement numbers and two (implied) GOTO's; it leads to unintelligible code, and is eschewed by good programmers. The Logical IF is better, in that the test part can be stated clearly, but hopelessly restrictive because the statement that follows the IF can only be one Fortran statement (with some *further* restrictions!). And of course there can be no ELSE part to a Fortran IF: there is no way to specify an alternative action if the IF is not satisfied.

The Fortran DO restricts the user to going forward in an arithmetic progression. It is fine for "1 to N in steps of 1 (or 2 or ...)", but there is no direct way to go backwards, or even (in ANSI Fortran[2]) to go from 1 to N−1. And of course the DO is useless if one's problem doesn't map into an arithmetic progression.

The result of these failings is that Fortran programs must be written with numerous labels and branches. The resulting code is particularly difficult to read and understand, and thus hard to debug and modify.

When one is faced with an unpleasant language, a useful technique is to define a new language that overcomes the deficiencies, and to translate it into the unpleasant one with a preprocessor. This is the approach taken with Ratfor. (The preprocessor idea is of course not new, and preprocessors for Fortran are especially popular today. A recent listing [3] of preprocessors shows more than 50, of which at least half a dozen are widely available.)

2. LANGUAGE DESCRIPTION

Design

Ratfor attempts to retain the merits of Fortran (universality, portability, efficiency) while hiding the worst Fortran inadequacies. The language *is* Fortran except for two aspects. First, since control flow is central to any program, regardless of the specific application, the primary task of Ratfor is to conceal this part of Fortran from the user, by providing decent control flow structures. These structures are sufficient and comfortable for structured programming in the narrow sense of programming without GOTO's. Second, since the preprocessor must examine an entire program to translate the control structure, it is possible at the same time to clean up many of the "cosmetic" deficiencies of Fortran, and thus provide a language which is easier and more pleasant to read and write.

Beyond these two aspects — control flow and cosmetics — Ratfor does nothing about the host of other weaknesses of Fortran. Although it would be straightforward to extend it to provide character strings, for example, they are not needed by everyone, and of course the preprocessor would be harder to implement. Throughout, the design principle which has determined what should be in Ratfor and what should not has been ***Ratfor doesn't know any Fortran.*** Any language feature which would require that Ratfor really understand Fortran has been omitted. We will return to this point in the section on implementation.

Even within the confines of control flow and cosmetics, we have attempted to be selective in

This paper is a revised and expanded version of oe published in *Software—Practice and Experience,* October 1975. The Ratfor described here is the one in use on UNIX and GCOS at Bell Laboratories, Murray Hill, N. J.

what features to provide. The intent has been to provide a small set of the most useful constructs, rather than to throw in everything that has ever been thought useful by someone.

The rest of this section contains an informal description of the Ratfor language. The control flow aspects will be quite familiar to readers used to languages like Algol, PL/I, Pascal, etc., and the cosmetic changes are equally straightforward. We shall concentrate on showing what the language looks like.

Statement Grouping

Fortran provides no way to group statements together, short of making them into a subroutine. The standard construction "if a condition is true, do this group of things," for example,

```
if (x > 100)
        { call error("x>100"); err = 1; return }
```

cannot be written directly in Fortran. Instead a programmer is forced to translate this relatively clear thought into murky Fortran, by stating the negative condition and branching around the group of statements:

```
        if (x .le. 100) goto 10
                call error(5hx>100)
                err = 1
                return
10      ...
```

When the program doesn't work, or when it must be modified, this must be translated back into a clearer form before one can be sure what it does.

Ratfor eliminates this error-prone and confusing back-and-forth translation; the first form *is* the way the computation is written in Ratfor. A group of statements can be treated as a unit by enclosing them in the braces { and }. This is true throughout the language: wherever a single Ratfor statement can be used, there can be several enclosed in braces. (Braces seem clearer and less obtrusive than **begin** and **end** or **do** and **end**, and of course **do** and **end** already have Fortran meanings.)

Cosmetics contribute to the readability of code, and thus to its understandability. The character ">" is clearer than ".GT.", so Ratfor translates it appropriately, along with several other similar shorthands. Although many Fortran compilers permit character strings in quotes (like **"x>100"**), quotes are not allowed in ANSI Fortran, so Ratfor converts it into the right number of **H**'s: computers count better than people do.

Ratfor is a free-form language: statements may appear anywhere on a line, and several may appear on one line if they are separated by semicolons. The example above could also be written as

```
if (x > 100) {
        call error("x>100")
        err = 1
        return
}
```

In this case, no semicolon is needed at the end of each line because Ratfor assumes there is one statement per line unless told otherwise.

Of course, if the statement that follows the **if** is a single statement (Ratfor or otherwise), no braces are needed:

```
if (y <= 0.0 & z <= 0.0)
        write(6, 20) y, z
```

No continuation need be indicated because the statement is clearly not finished on the first line. In general Ratfor continues lines when it seems obvious that they are not yet done. (The continuation convention is discussed in detail later.)

Although a free-form language permits wide latitude in formatting styles, it is wise to pick one that is readable, then stick to it. In particular, proper indentation is vital, to make the logical structure of the program obvious to the reader.

The "else" Clause

Ratfor provides an **else** statement to handle the construction "if a condition is true, do this thing, *otherwise* do that thing."

```
if (a <= b)
        { sw = 0; write(6, 1) a, b }
else
        { sw = 1; write(6, 1) b, a }
```

This writes out the smaller of **a** and **b**, then the larger, and sets **sw** appropriately.

The Fortran equivalent of this code is circuitous indeed:

```
        if (a .gt. b) goto 10
                sw = 0
                write(6, 1) a, b
                goto 20
10      sw = 1
        write(6, 1) b, a
20      ...
```

This is a mechanical translation; shorter forms exist, as they do for many similar situations. But all translations suffer from the same problem: since they are translations, they are less clear and understandable than code that is not a translation. To understand the Fortran version, one must scan the entire program to make sure that no other statement branches to statements 10 or 20 before one knows that indeed this is an **if-else** construc-

tion. With the Ratfor version, there is no question about how one gets to the parts of the statement. The **if-else** is a single unit, which can be read, understood, and ignored if not relevant. The program says what it means.

As before, if the statement following an **if** or an **else** is a single statement, no braces are needed:

```
if (a <= b)
        sw = 0
else
        sw = 1
```

The syntax of the **if** statement is

```
if (legal Fortran condition)
        Ratfor statement
else
        Ratfor statement
```

where the **else** part is optional. The *legal Fortran condition* is anything that can legally go into a Fortran Logical IF. Ratfor does not check this clause, since it does not know enough Fortran to know what is permitted. The *Ratfor statement* is any Ratfor or Fortran statement, or any collection of them in braces.

Nested if's

Since the statement that follows an **if** or an **else** can be any Ratfor statement, this leads immediately to the possibility of another **if** or **else**. As a useful example, consider this problem: the variable **f** is to be set to −1 if **x** is less than zero, to +1 if **x** is greater than 100, and to 0 otherwise. Then in Ratfor, we write

```
if (x < 0)
        f = -1
else if (x > 100)
        f = +1
else
        f = 0
```

Here the statement after the first **else** is another **if-else**. Logically it is just a single statement, although it is rather complicated.

This code says what it means. Any version written in straight Fortran will necessarily be indirect because Fortran does not let you say what you mean. And as always, clever shortcuts may turn out to be too clever to understand a year from now.

Following an **else** with an **if** is one way to write a multi-way branch in Ratfor. In general the structure

```
if (...)
        - - -
else if (...)
        - - -
else if (...)
        - - -
...
else
        - - -
```

provides a way to specify the choice of exactly one of several alternatives. (Ratfor also provides a **switch** statement which does the same job in certain special cases; in more general situations, we have to make do with spare parts.) The tests are laid out in sequence, and each one is followed by the code associated with it. Read down the list of decisions until one is found that is satisfied. The code associated with this condition is executed, and then the entire structure is finished. The trailing **else** part handles the "default" case, where none of the other conditions apply. If there is no default action, this final **else** part is omitted:

```
if (x < 0)
        x = 0
else if (x > 100)
        x = 100
```

if-else ambiguity

There is one thing to notice about complicated structures involving nested **if**'s and **else**'s. Consider

```
if (x > 0)
        if (y > 0)
                write(6, 1) x, y
        else
                write(6, 2) y
```

There are two **if**'s and only one **else**. Which **if** does the **else** go with?

This is a genuine ambiguity in Ratfor, as it is in many other programming languages. The ambiguity is resolved in Ratfor (as elsewhere) by saying that in such cases the **else** goes with the closest previous un-**else**'ed **if**. Thus in this case, the **else** goes with the inner **if**, as we have indicated by the indentation.

It is a wise practice to resolve such cases by explicit braces, just to make your intent clear. In the case above, we would write

```
if (x > 0) {
        if (y > 0)
                write(6, 1) x, y
        else
                write(6, 2) y
}
```

which does not change the meaning, but leaves no doubt in the reader's mind. If we want the other association, we *must* write

```
if (x > 0) {
        if (y > 0)
                write(6, 1) x, y
}
else
        write(6, 2) y
```

The "switch" Statement

The **switch** statement provides a clean way to express multi-way branches which branch on the value of some integer-valued expression. The syntax is

```
switch (expression) {

        case expr1 :
                statements
        case expr2, expr3 :
                statements
        ...
        default:
                statements
}
```

Each **case** is followed by a list of comma-separated integer expressions. The *expression* inside **switch** is compared against the case expressions *expr1, expr2,* and so on in turn until one matches, at which time the statements following that **case** are executed. If no cases match *expression,* and there is a **default** section, the statements with it are done; if there is no **default,** nothing is done. In all situations, as soon as some block of statements is executed, the entire **switch** is exited immediately. (Readers familiar with C[4] should beware that this behavior is not the same as the C **switch**.)

The "do" Statement

The **do** statement in Ratfor is quite similar to the DO statement in Fortran, except that it uses no statement number. The statement number, after all, serves only to mark the end of the DO, and this can be done just as easily with braces. Thus

```
do i = 1, n {
        x(i) = 0.0
        y(i) = 0.0
        z(i) = 0.0
}
```

is the same as

```
do 10 i = 1, n
        x(i) = 0.0
        y(i) = 0.0
        z(i) = 0.0
10      continue
```

The syntax is:

```
do legal-Fortran-DO-text
        Ratfor statement
```

The part that follows the keyword **do** has to be something that can legally go into a Fortran DO statement. Thus if a local version of Fortran allows DO limits to be expressions (which is not currently permitted in ANSI Fortran), they can be used in a Ratfor **do.**

The *Ratfor statement* part will often be enclosed in braces, but as with the **if**, a single statement need not have braces around it. This code sets an array to zero:

```
do i = 1, n
        x(i) = 0.0
```

Slightly more complicated,

```
do i = 1, n
        do j = 1, n
                m(i, j) = 0
```

sets the entire array **m** to zero, and

```
do i = 1, n
        do j = 1, n
                if (i < j)
                        m(i, j) = -1
                else if (i == j)
                        m(i, j) = 0
                else
                        m(i, j) = +1
```

sets the upper triangle of **m** to –1, the diagonal to zero, and the lower triangle to +1. (The operator == is "equals", that is, ".EQ.".) In each case, the statement that follows the **do** is logically a *single* statement, even though complicated, and thus needs no braces.

"break" and "next"

Ratfor provides a statement for leaving a loop early, and one for beginning the next iteration. **break** causes an immediate exit from the **do**; in effect it is a branch to the statement *after* the **do**. **next** is a branch to the bottom of the loop, so it causes the next iteration to be done. For example, this code skips over negative values in an array:

```
do i = 1, n {
        if (x(i) < 0.0)
                next
        process positive element
}
```

break and **next** also work in the other Ratfor looping constructions that we will talk about in the next few sections.

break and **next** can be followed by an integer to indicate breaking or iterating that level of enclosing loop; thus

```
break 2
```

exits from two levels of enclosing loops, and **break 1** is equivalent to **break**. **next 2** iterates the second enclosing loop. (Realistically, multi-level **break**'s and **next**'s are not likely to be much used because they lead to code that is hard to understand and somewhat risky to change.)

The "while" Statement

One of the problems with the Fortran DO statement is that it generally insists upon being done once, regardless of its limits. If a loop begins

```
DO I = 2, 1
```

this will typically be done once with **I** set to 2, even though common sense would suggest that perhaps it shouldn't be. Of course a Ratfor **do** can easily be preceded by a test

```
if (j <= k)
        do i = j, k {
                ---
        }
```

but this has to be a conscious act, and is often overlooked by programmers.

A more serious problem with the DO statement is that it encourages that a program be written in terms of an arithmetic progression with small positive steps, even though that may not be the best way to write it. If code has to be contorted to fit the requirements imposed by the Fortran DO, it is that much harder to write and understand.

To overcome these difficulties, Ratfor provides a **while** statement, which is simply a loop: "while some condition is true, repeat this group of statements". It has no preconceptions about why one is looping. For example, this routine to compute sin(x) by the Maclaurin series combines two termination criteria.

```
real function sin(x, e)
    # returns sin(x) to accuracy e, by
    # sin(x) = x - x**3/3! + x**5/5! - ...

    sin = x
    term = x

    i = 3
    while (abs(term)>e & i<100) {
        term = -term * x**2 / float(i*(i-1))
        sin = sin + term
        i = i + 2
    }

    return
    end
```

Notice that if the routine is entered with **term** already smaller than **e**, the loop will be done *zero times*, that is, no attempt will be made to compute **x******3** and thus a potential underflow is avoided. Since the test is made at the top of a **while** loop instead of the bottom, a special case disappears — the code works at one of its boundaries. (The test **i<100** is the other boundary — making sure the routine stops after some maximum number of iterations.)

As an aside, a sharp character "#" in a line marks the beginning of a comment; the rest of the line is comment. Comments and code can co-exist on the same line — one can make marginal remarks, which is not possible with Fortran's "C in column 1" convention. Blank lines are also permitted anywhere (they are not in Fortran); they should be used to emphasize the natural divisions of a program.

The syntax of the **while** statement is

```
while (legal Fortran condition)
        Ratfor statement
```

As with the **if**, *legal Fortran condition* is something that can go into a Fortran Logical IF, and *Ratfor statement* is a single statement, which may be multiple statements in braces.

The **while** encourages a style of coding not normally practiced by Fortran programmers. For example, suppose **nextch** is a function which returns the next input character both as a function value and in its argument. Then a loop to find the first non-blank character is just

```
while (nextch(ich) == iblank)
        ;
```

A semicolon by itself is a null statement, which is necessary here to mark the end of the **while**; if it were not present, the **while** would control the next statement. When the loop is broken, **ich** contains the first non-blank. Of course the same code can

be written in Fortran as

```
100     if (nextch(ich) .eq. iblank) goto 100
```

but many Fortran programmers (and a few compilers) believe this line is illegal. The language at one's disposal strongly influences how one thinks about a problem.

The "for" Statement

The **for** statement is another Ratfor loop, which attempts to carry the separation of loop-body from reason-for-looping a step further than the **while.** A **for** statement allows explicit initialization and increment steps as part of the statement. For example, a DO loop is just

```
for (i = 1; i <= n; i = i + 1) ...
```

This is equivalent to

```
i = 1
while (i <= n) {
        ...
        i = i + 1
}
```

The initialization and increment of **i** have been moved into the **for** statement, making it easier to see at a glance what controls the loop.

The **for** and **while** versions have the advantage that they will be done zero times if **n** is less than 1; this is not true of the **do**.

The loop of the sine routine in the previous section can be re-written with a **for** as

```
for (i=3; abs(term) > e & i < 100; i=i+2) {
        term = -term * x**2 / float(i*(i-1))
        sin = sin + term
}
```

The syntax of the **for** statement is

```
for ( init ; condition ; increment )
        Ratfor statement
```

init is any single Fortran statement, which gets done once before the loop begins. *increment* is any single Fortran statement, which gets done at the end of each pass through the loop, before the test. *condition* is again anything that is legal in a logical IF. Any of *init, condition,* and *increment* may be omitted, although the semicolons *must* always be present. A non-existent *condition* is treated as always true, so **for(;;)** is an indefinite repeat. (But see the **repeat-until** in the next section.)

The **for** statement is particularly useful for backward loops, chaining along lists, loops that might be done zero times, and similar things which are hard to express with a DO statement, and obscure to write out with IF's and GOTO's. For example, here is a backwards DO loop to find the last non-blank character on a card:

```
for (i = 80; i > 0; i = i - 1)
        if (card(i) != blank)
                break
```

("!=" is the same as ".NE."). The code scans the columns from 80 through to 1. If a non-blank is found, the loop is immediately broken. (**break** and **next** work in **for**'s and **while**'s just as in **do**'s). If **i** reaches zero, the card is all blank.

This code is rather nasty to write with a regular Fortran DO, since the loop must go forward, and we must explicitly set up proper conditions when we fall out of the loop. (Forgetting this is a common error.) Thus:

```
      DO 10 J = 1, 80
           I = 81 - J
           IF (CARD(I) .NE. BLANK) GO TO 11
10    CONTINUE
      I = 0
11    ...
```

The version that uses the **for** handles the termination condition properly for free; **i** *is* zero when we fall out of the **for** loop.

The increment in a **for** need not be an arithmetic progression; the following program walks along a list (stored in an integer array **ptr**) until a zero pointer is found, adding up elements from a parallel array of values:

```
sum = 0.0
for (i = first; i > 0; i = ptr(i))
        sum = sum + value(i)
```

Notice that the code works correctly if the list is empty. Again, placing the test at the top of a loop instead of the bottom eliminates a potential boundary error.

The "repeat-until" statement

In spite of the dire warnings, there are times when one really needs a loop that tests at the bottom after one pass through. This service is provided by the **repeat-until**:

```
repeat
        Ratfor statement
until (legal Fortran condition)
```

The *Ratfor statement* part is done once, then the condition is evaluated. If it is true, the loop is exited; if it is false, another pass is made.

The **until** part is optional, so a bare **repeat** is the cleanest way to specify an infinite loop. Of course such a loop must ultimately be broken by some transfer of control such as **stop**, **return**, or **break**, or an implicit stop such as running out of input with a READ statement.

As a matter of observed fact[8], the **repeat-until** statement is *much* less used than the other looping constructions; in particular, it is typically outnumbered ten to one by **for** and **while**. Be cautious about using it, for loops that test only at the bottom often don't handle null cases well.

More on break and next

break exits immediately from **do**, **while**, **for**, and **repeat-until**. **next** goes to the test part of **do**, **while** and **repeat-until**, and to the increment step of a **for**.

"return" Statement

The standard Fortran mechanism for returning a value from a function uses the name of the function as a variable which can be assigned to; the last value stored in it is the function value upon return. For example, here is a routine **equal** which returns 1 if two arrays are identical, and zero if they differ. The array ends are marked by the special value –1.

```
# equal _ compare str1 to str2;
#     return 1 if equal, 0 if not
      integer function equal(str1, str2)
      integer str1(100), str2(100)
      integer i

      for (i = 1; str1(i) == str2(i); i = i + 1)
            if (str1(i) == -1) {
                  equal = 1
                  return
            }
      equal = 0
      return
      end
```

In many languages (e.g., PL/I) one instead says

return (*expression*)

to return a value from a function. Since this is often clearer, Ratfor provides such a **return** statement — in a function **F**, **return**(expression) is equivalent to

{ F = expression; return }

For example, here is **equal** again:

```
# equal _ compare str1 to str2;
#     return 1 if equal, 0 if not
      integer function equal(str1, str2)
      integer str1(100), str2(100)
      integer i

      for (i = 1; str1(i) == str2(i); i = i + 1)
            if (str1(i) == -1)
                  return(1)
      return(0)
      end
```

If there is no parenthesized expression after **return**, a normal RETURN is made. (Another version of **equal** is presented shortly.)

Cosmetics

As we said above, the visual appearance of a language has a substantial effect on how easy it is to read and understand programs. Accordingly, Ratfor provides a number of cosmetic facilities which may be used to make programs more readable.

Free-form Input

Statements can be placed anywhere on a line; long statements are continued automatically, as are long conditions in **if**, **while**, **for**, and **until**. Blank lines are ignored. Multiple statements may appear on one line, if they are separated by semicolons. No semicolon is needed at the end of a line, if Ratfor can make some reasonable guess about whether the statement ends there. Lines ending with any of the characters

= + - * , | & (_

are assumed to be continued on the next line. Underscores are discarded wherever they occur; all others remain as part of the statement.

Any statement that begins with an all-numeric field is assumed to be a Fortran label, and placed in columns 1-5 upon output. Thus

```
write(6, 100); 100 format("hello")
```

is converted into

```
      write(6, 100)
100   format(5hhello)
```

Translation Services

Text enclosed in matching single or double quotes is converted to **nH...** but is otherwise unaltered (except for formatting — it may get split across card boundaries during the reformatting process). Within quoted strings, the backslash '\' serves as an escape character: the next character is taken literally. This provides a way to get quotes

(and of course the backslash itself) into quoted strings:

```
"\\'"
```

is a string containing a backslash and an apostrophe. (This is *not* the standard convention of doubled quotes, but it is easier to use and more general.)

Any line that begins with the character '%' is left absolutely unaltered except for stripping off the '%' and moving the line one position to the left. This is useful for inserting control cards, and other things that should not be transmogrified (like an existing Fortran program). Use '%' only for ordinary statements, not for the condition parts of **if**, **while**, etc., or the output may come out in an unexpected place.

The following character translations are made, except within single or double quotes or on a line beginning with a '%'.

==	.eq.	!=	.ne.
>	.gt.	>=	.ge.
<	.lt.	<=	.le.
&	.and.	|	.or.
!	.not.	¬	.not.

In addition, the following translations are provided for input devices with restricted character sets.

[	{	]	}
$(	{	$)	}

"define" Statement

Any string of alphanumeric characters can be defined as a name; thereafter, whenever that name occurs in the input (delimited by non-alphanumerics) it is replaced by the rest of the definition line. (Comments and trailing white spaces are stripped off). A defined name can be arbitrarily long, and must begin with a letter.

define is typically used to create symbolic parameters:

```
define ROWS 100
define COLS 50

dimension a(ROWS), b(ROWS, COLS)

        if (i > ROWS | j > COLS) ...
```

Alternately, definitions may be written as

```
define(ROWS, 100)
```

In this case, the defining text is everything after the comma up to the balancing right parenthesis; this allows multi-line definitions.

It is generally a wise practice to use symbolic parameters for most constants, to help make clear the function of what would otherwise be mysterious numbers. As an example, here is the routine **equal** again, this time with symbolic constants.

```
define    YES     1
define    NO      0
define    EOS     -1
define    ARB     100

# equal _ compare str1 to str2;
#     return YES if equal, NO if not
      integer function equal(str1, str2)
      integer str1(ARB), str2(ARB)
      integer i

      for (i = 1; str1(i) == str2(i); i = i + 1)
            if (str1(i) == EOS)
                  return(YES)
      return(NO)
      end
```

"include" Statement

The statement

```
include file
```

inserts the file found on input stream *file* into the Ratfor input in place of the **include** statement. The standard usage is to place COMMON blocks on a file, and **include** that file whenever a copy is needed:

```
subroutine x
      include commonblocks
      ...
      end

suroutine y
      include commonblocks
      ...
      end
```

This ensures that all copies of the COMMON blocks are identical

Pitfalls, Botches, Blemishes and other Failings

Ratfor catches certain syntax errors, such as missing braces, **else** clauses without an **if**, and most errors involving missing parentheses in statements. Beyond that, since Ratfor knows no Fortran, any errors you make will be reported by the Fortran compiler, so you will from time to time have to relate a Fortran diagnostic back to the Ratfor source.

Keywords are reserved — using **if**, **else**, etc., as variable names will typically wreak havoc. Don't leave spaces in keywords. Don't use the Arithmetic IF.

The Fortran **nH** convention is not recognized anywhere by Ratfor; use quotes instead.

3. IMPLEMENTATION

Ratfor was originally written in C[4] on the UNIX operating system[5]. The language is specified by a context free grammar and the compiler constructed using the YACC compiler-compiler[6].

The Ratfor grammar is simple and straightforward, being essentially

```
prog    : stat
        | prog  stat
stat    : if (...) stat
        | if (...) stat else stat
        | while (...) stat
        | for (...; ...; ...) stat
        | do ... stat
        | repeat stat
        | repeat stat until (...)
        | switch (...) { case ...: prog ...
                         default: prog }
        | return
        | break
        | next
        | digits  stat
        | { prog }
        | anything unrecognizable
```

The observation that Ratfor knows no Fortran follows directly from the rule that says a statement is "anything unrecognizable". In fact most of Fortran falls into this category, since any statement that does not begin with one of the keywords is by definition "unrecognizable."

Code generation is also simple. If the first thing on a source line is not a keyword (like **if**, **else**, etc.) the entire statement is simply copied to the output with appropriate character translation and formatting. (Leading digits are treated as a label.) Keywords cause only slightly more complicated actions. For example, when **if** is recognized, two consecutive labels L and L+1 are generated and the value of L is stacked. The condition is then isolated, and the code

```
if (.not. (condition)) goto L
```

is output. The *statement* part of the **if** is then translated. When the end of the statement is encountered (which may be some distance away and include nested **if**'s, of course), the code

```
L       continue
```

is generated, unless there is an **else** clause, in which case the code is

```
        goto L+1
L       continue
```

In this latter case, the code

```
L+1     continue
```

is produced after the *statement* part of the **else**. Code generation for the various loops is equally simple.

One might argue that more care should be taken in code generation. For example, if there is no trailing **else**,

```
if (i > 0) x = a
```

should be left alone, not converted into

```
        if (.not. (i .gt. 0)) goto 100
        x = a
100     continue
```

But what are optimizing compilers for, if not to improve code? It is a rare program indeed where this kind of "inefficiency" will make even a measurable difference. In the few cases where it is important, the offending lines can be protected by '%'.

The use of a compiler-compiler is definitely the preferred method of software development. The language is well-defined, with few syntactic irregularities. Implementation is quite simple; the original construction took under a week. The language is sufficiently simple, however, that an *ad hoc* recognizer can be readily constructed to do the same job if no compiler-compiler is available.

The C version of Ratfor is used on UNIX and on the Honeywell GCOS systems. C compilers are not as widely available as Fortran, however, so there is also a Ratfor written in itself and originally bootstrapped with the C version. The Ratfor version was written so as to translate into the portable subset of Fortran described in [1], so it is portable, having been run essentially without change on at least twelve distinct machines. (The main restrictions of the portable subset are: only one character per machine word; subscripts in the form $c*v\pm c$; avoiding expressions in places like DO loops; consistency in subroutine argument usage, and in COMMON declarations. Ratfor itself will not gratuitously generate non-standard Fortran.)

The Ratfor version is about 1500 lines of Ratfor (compared to about 1000 lines of C); this compiles into 2500 lines of Fortran. This expansion ratio is somewhat higher than average, since the compiled code contains unnecessary occurrences of COMMON declarations. The execution time of the Ratfor version is dominated by two routines that read and write cards. Clearly these routines could be replaced by machine coded local versions; unless this is done, the efficiency of other parts of the translation process is largely irrelevant.

4. EXPERIENCE

Good Things

"It's so much better than Fortran" is the most common response of users when asked how well Ratfor meets their needs. Although cynics might consider this to be vacuous, it does seem to be true that decent control flow and cosmetics converts Fortran from a bad language into quite a reasonable one, assuming that Fortran data structures are adequate for the task at hand.

Although there are no quantitative results, users feel that coding in Ratfor is at least twice as fast as in Fortran. More important, debugging and subsequent revision are much faster than in Fortran. Partly this is simply because the code can be *read*. The looping statements which test at the top instead of the bottom seem to eliminate or at least reduce the occurrence of a wide class of boundary errors. And of course it is easy to do structured programming in Ratfor; this self-discipline also contributes markedly to reliability.

One interesting and encouraging fact is that programs written in Ratfor tend to be as readable as programs written in more modern languages like Pascal. Once one is freed from the shackles of Fortran's clerical detail and rigid input format, it is easy to write code that is readable, even esthetically pleasing. For example, here is a Ratfor implementation of the linear table search discussed by Knuth [7]:

```
A(m+1) = x
for (i = 1; A(i) != x; i = i + 1)
        ;
if (i > m) {
        m = i
        B(i) = 1
}
else
        B(i) = B(i) + 1
```

A large corpus (5400 lines) of Ratfor, including a subset of the Ratfor preprocessor itself, can be found in [8].

Bad Things

The biggest single problem is that many Fortran syntax errors are not detected by Ratfor but by the local Fortran compiler. The compiler then prints a message in terms of the generated Fortran, and in a few cases this may be difficult to relate back to the offending Ratfor line, especially if the implementation conceals the generated Fortran. This problem could be dealt with by tagging each generated line with some indication of the source line that created it, but this is inherently implementation-dependent, so no action has yet been taken. Error message interpretation is actually not so arduous as might be thought. Since Ratfor generates no variables, only a simple pattern of IF's and GOTO's, data-related errors like missing DIMENSION statements are easy to find in the Fortran. Furthermore, there has been a steady improvement in Ratfor's ability to catch trivial syntactic errors like unbalanced parentheses and quotes.

There are a number of implementation weaknesses that are a nuisance, especially to new users. For example, keywords are reserved. This rarely makes any difference, except for those hardy souls who want to use an Arithmetic IF. A few standard Fortran constructions are not accepted by Ratfor, and this is perceived as a problem by users with a large corpus of existing Fortran programs. Protecting every line with a '%' is not really a complete solution, although it serves as a stop-gap. The best long-term solution is provided by the program Struct [9], which converts arbitrary Fortran programs into Ratfor.

Users who export programs often complain that the generated Fortran is "unreadable" because it is not tastefully formatted and contains extraneous CONTINUE statements. To some extent this can be ameliorated (Ratfor now has an option to copy Ratfor comments into the generated Fortran), but it has always seemed that effort is better spent on the input language than on the output esthetics.

One final problem is partly attributable to success — since Ratfor is relatively easy to modify, there are now several dialects of Ratfor. Fortunately, so far most of the differences are in character set, or in invisible aspects like code generation.

5. CONCLUSIONS

Ratfor demonstrates that with modest effort it is possible to convert Fortran from a bad language into quite a good one. A preprocessor is clearly a useful way to extend or ameliorate the facilities of a base language.

When designing a language, it is important to concentrate on the essential requirement of providing the user with the best language possible for a given effort. One must avoid throwing in "features" — things which the user may trivially construct within the existing framework.

One must also avoid getting sidetracked on irrelevancies. For instance it seems pointless for Ratfor to prepare a neatly formatted listing of either its input or its output. The user is presumably capable of the self-discipline required to prepare neat input that reflects his thoughts. It is much more important that the language provide free-form input so he *can* format it neatly. No one

should read the output anyway except in the most dire circumstances.

Acknowledgements

C. A. R. Hoare once said that "One thing [the language designer] should not do is to include untried ideas of his own." Ratfor follows this precept very closely — everything in it has been stolen from someone else. Most of the control flow structures are taken directly from the language C[4] developed by Dennis Ritchie; the comment and continuation conventions are adapted from Altran[10].

I am grateful to Stuart Feldman, whose patient simulation of an innocent user during the early days of Ratfor led to several design improvements and the eradication of bugs. He also translated the C parse-tables and YACC parser into Fortran for the first Ratfor version of Ratfor.

References

[1] B. G. Ryder, "The PFORT Verifier," *Software—Practice & Experience,* October 1974.

[2] American National Standard Fortran. American National Standards Institute, New York, 1966.

[3] *For-word: Fortran Development Newsletter,* August 1975.

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

[5] D. M. Ritchie and K. L. Thompson, "The UNIX Time-sharing System." *CACM,* July 1974.

[6] S. C. Johnson, "YACC — Yet Another Compiler-Compiler." Bell Laboratories Computing Science Technical Report #32, 1978.

[7] D. E. Knuth, "Structured Programming with goto Statements." *Computing Surveys,* December 1974.

[8] B. W. Kernighan and P. J. Plauger, *Software Tools,* Addison-Wesley, 1976.

[9] B. S. Baker, "Struct — A Program which Structures Fortran", Bell Laboratories internal memorandum, December 1975.

[10] A. D. Hall, "The Altran System for Rational Function Manipulation — A Survey." *CACM,* August 1971.

Appendix: Usage on UNIX and GCOS.

Beware — local customs vary. Check with a native before going into the jungle.

UNIX

The program **ratfor** is the basic translator; it takes either a list of file names or the standard input and writes Fortran on the standard output. Options include **–6x**, which uses **x** as a continuation character in column 6 (UNIX uses **&** in column 1), and **–C**, which causes Ratfor comments to be copied into the generated Fortran.

The program **rc** provides an interface to the **ratfor** command which is much the same as **cc**. Thus

rc [options] files

compiles the files specified by **files**. Files with names ending in **.r** are Ratfor source; other files are assumed to be for the loader. The flags **–C** and **–6x** described above are recognized, as are

–c	compile only; don′ t load
–f	save intermediate Fortran .f files
–r	Ratfor only; implies –c and –f
–2	use big Fortran compiler (for large programs)
–U	flag undeclared variables (not universally available)

Other flags are passed on to the loader.

GCOS

The program **./ratfor** is the bare translator, and is identical to the UNIX version, except that the continuation convention is & in column 6. Thus

./ratfor files >output

translates the Ratfor source on **files** and collects the generated Fortran on file 'output' for subsequent processing.

./rc provides much the same services as **rc** (within the limitations of GCOS), regrettably with a somewhat different syntax. Options recognized by **./rc** include

name	Ratfor source or library, depending on type
h=/name	make TSS H* file (runnable version); run as /name
r=/name	update and use random library
a=	compile as ascii (default is bcd)
C=	copy comments into Fortran
f=name	Fortran source file
g=name	gmap source file

Other options are as specified for the **./cc** command described in [4].

TSO, TSS, and other systems

Ratfor exists on various other systems; check with the author for specifics.

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.

July 1, 1977

† UNIX is a trademark of Bell Laboratories.

The M4 Macro Processor

Brian W. Kernighan

Dennis M. Ritchie

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 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 subsequent **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.

SED — A Non-interactive Text Editor

Lee E. McMahon

Context search
Editing

ABSTRACT

Sed is a non-interactive context editor that runs on the UNIX† operating system. *Sed* is designed to be especially useful in three cases:

1) To edit files too large for comfortable interactive editing;
2) To edit any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode.
3) To perform multiple 'global' editing functions efficiently in one pass through the input.

This memorandum constitutes a manual for users of *sed*.

August 15, 1978

† UNIX is a trademark of Bell Laboratories.

SED — A Non-interactive Text Editor

Lee E. McMahon

Context search
Editing

Introduction

Sed is a non-interactive context editor designed to be especially useful in three cases:

1) To edit files too large for comfortable interactive editing;
2) To edit any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode;
3) To perform multiple 'global' editing functions efficiently in one pass through the input.

Since only a few lines of the input reside in core at one time, and no temporary files are used, the effective size of file that can be edited is limited only by the requirement that the input and output fit simultaneously into available secondary storage.

Complicated editing scripts can be created separately and given to *sed* as a command file. For complex edits, this saves considerable typing, and its attendant errors. *Sed* running from a command file is much more efficient than any interactive editor known to the author, even if that editor can be driven by a pre-written script.

The principal loss of functions compared to an interactive editor are lack of relative addressing (because of the line-at-a-time operation), and lack of immediate verification that a command has done what was intended.

Sed is a lineal descendant of the UNIX editor, *ed*. Because of the differences between interactive and non-interactive operation, considerable changes have been made between *ed* and *sed;* even confirmed users of *ed* will frequently be surprised (and probably chagrined), if they rashly use *sed* without reading Sections 2 and 3 of this document. The most striking family resemblance between the two editors is in the class of patterns ('regular expressions') they recognize; the code for matching patterns is copied almost verbatim from the code for *ed,* and the description of regular expressions in Section 2 is copied almost verbatim from the UNIX Programmer's Manual[1]. (Both code and description were written by Dennis M. Ritchie.)

1. Overall Operation

Sed by default copies the standard input to the standard output, perhaps performing one or more editing commands on each line before writing it to the output. This behavior may be modified by flags on the command line; see Section 1.1 below.

The general format of an editing command is:

[address1,address2][function][arguments]

One or both addresses may be omitted; the format of addresses is given in Section 2. Any number of blanks or tabs may separate the addresses from the function. The function must be present; the available commands are discussed in Section 3. The arguments may be required or optional, according to which function is given; again, they are discussed in Section 3 under each individual function.

Tab characters and spaces at the beginning of lines are ignored.

1.1. Command-line Flags

Three flags are recognized on the command line:

-n: tells *sed* not to copy all lines, but only those specified by *p* functions or *p* flags after *s* functions (see Section 3.3);

-e: tells *sed* to take the next argument as an editing command;

-f: tells *sed* to take the next argument as a file name; the file should contain editing commands, one to a line.

1.2. Order of Application of Editing Commands

Before any editing is done (in fact, before any input file is even opened), all the editing commands are compiled into a form which will be moderately efficient during the execution phase (when the commands are actually applied to lines of the input file). The commands are compiled in the order in which they are encountered; this is generally the order in which they will be attempted at execution time. The commands are applied one at a time; the input to each command is the output of all preceding commands.

The default linear order of application of editing commands can be changed by the flow-of-control commands, *t* and *b* (see Section 3). Even when the order of application is changed by these commands, it is still true that the input line to any command is the output of any previously applied command.

1.3. Pattern-space

The range of pattern matches is called the pattern space. Ordinarily, the pattern space is one line of the input text, but more than one line can be read into the pattern space by using the *N* command (Section 3.6.).

1.4. Examples

Examples are scattered throughout the text. Except where otherwise noted, the examples all assume the following input text:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

(In no case is the output of the *sed* commands to be considered an improvement on Coleridge.)

Example:

The command

```
2q
```

will quit after copying the first two lines of the input. The output will be:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
```

2. ADDRESSES: Selecting lines for editing

Lines in the input file(s) to which editing commands are to be applied can be selected by addresses. Addresses may be either line numbers or context addresses.

The application of a group of commands can be controlled by one address (or address-pair) by grouping the commands with curly braces ('{ }')(Sec. 3.6.).

2.1. Line-number Addresses

A line number is a decimal integer. As each line is read from the input, a line-number counter is incremented; a line-number address matches (selects) the input line which causes the internal counter to equal the address line-number. The counter runs cumulatively through multiple input files; it is not reset when a new input file is opened.

As a special case, the character **$** matches the last line of the last input file.

2.2. Context Addresses

A context address is a pattern ('regular expression') enclosed in slashes ('/'). The regular expressions recognized by *sed* are constructed as follows:

1) An ordinary character (not one of those discussed below) is a regular expression, and matches that character.
2) A circumflex '^' at the beginning of a regular expression matches the null character at the beginning of a line.
3) A dollar-sign '$' at the end of a regular expression matches the null character at the end of a line.
4) The characters '\n' match an imbedded newline character, but not the newline at the end of the pattern space.
5) A period '.' matches any character except the terminal newline of the pattern space.
6) A regular expression followed by an asterisk '*' matches any number (including 0) of adjacent occurrences of the regular expression it follows.
7) A string of characters in square brackets '[ ]' matches any character in the string, and no others. If, however, the first character of the string is circumflex '^', the regular expression matches any character *except* the characters in the string and the terminal newline of the pattern space.
8) A concatenation of regular expressions is a regular expression which matches the concatenation of strings matched by the components of the regular expression.
9) A regular expression between the sequences '\(' and '\)' is identical in effect to the unadorned regular expression, but has side-effects which are described under the *s* command below and specification 10) immediately below.
10) The expression '*d*' means the same string of characters matched by an expression enclosed in '\(' and '\)' earlier in the same pattern. Here *d* is a single digit; the string specified is that beginning with the *d*th occurrence of '\(' counting from the left. For example, the expression '^\(.*\)\1' matches a line beginning with two repeated occurrences of the same string.
11) The null regular expression standing alone (e.g., '//') is equivalent to the last regular expression compiled.

To use one of the special characters (^ $. * [ ] \ /) as a literal (to match an occurrence of itself in the input), precede the special character by a backslash '\'.

For a context address to 'match' the input requires that the whole pattern within the address match some portion of the pattern space.

2.3. Number of Addresses

The commands in the next section can have 0, 1, or 2 addresses. Under each command the maximum number of allowed addresses is given. For a command to have more addresses than the maximum allowed is considered an error.

If a command has no addresses, it is applied to every line in the input.

If a command has one address, it is applied to all lines which match that address.

If a command has two addresses, it is applied to the first line which matches the first address, and to all subsequent lines until (and including) the first subsequent line which matches the second address. Then an attempt is made on subsequent lines to again match the first address, and the

process is repeated.

Two addresses are separated by a comma.

Examples:

/an/	matches lines 1, 3, 4 in our sample text
/an.*an/	matches line 1
/^an/	matches no lines
/./	matches all lines
/\./	matches line 5
/r*an/	matches lines 1,3, 4 (number = zero!)
/\(an\).*\1/	matches line 1

3. FUNCTIONS

All functions are named by a single character. In the following summary, the maximum number of allowable addresses is given enclosed in parentheses, then the single character function name, possible arguments enclosed in angles (< >), an expanded English translation of the single-character name, and finally a description of what each function does. The angles around the arguments are *not* part of the argument, and should not be typed in actual editing commands.

3.1. Whole-line Oriented Functions

(2)d -- delete lines

The *d* function deletes from the file (does not write to the output) all those lines matched by its address(es).

It also has the side effect that no further commands are attempted on the corpse of a deleted line; as soon as the *d* function is executed, a new line is read from the input, and the list of editing commands is re-started from the beginning on the new line.

(2)n -- next line

The *n* function reads the next line from the input, replacing the current line. The current line is written to the output if it should be. The list of editing commands is continued following the *n* command.

(1)a\
<text> -- append lines

The *a* function causes the argument <text> to be written to the output after the line matched by its address. The *a* command is inherently multi-line; *a* must appear at the end of a line, and <text> may contain any number of lines. To preserve the one-command-to-a-line fiction, the interior newlines must be hidden by a backslash character ('\') immediately preceding the newline. The <text> argument is terminated by the first unhidden newline (the first one not immediately preceded by backslash).

Once an *a* function is successfully executed, <text> will be written to the output regardless of what later commands do to the line which triggered it. The triggering line may be deleted entirely; <text> will still be written to the output.

The <text> is not scanned for address matches, and no editing commands are attempted on it. It does not cause any change in the line-number counter.

(1)i\
<text> -- insert lines

The *i* function behaves identically to the *a* function, except that <text> is

written to the output *before* the matched line. All other comments about the *a* function apply to the *i* function as well.

(2)c\
<text> -- change lines

The *c* function deletes the lines selected by its address(es), and replaces them with the lines in <text>. Like *a* and *i*, *c* must be followed by a newline hidden by a backslash; and interior new lines in <text> must be hidden by backslashes.

The *c* command may have two addresses, and therefore select a range of lines. If it does, all the lines in the range are deleted, but only one copy of <text> is written to the output, *not* one copy per line deleted. As with *a* and *i*, <text> is not scanned for address matches, and no editing commands are attempted on it. It does not change the line-number counter.

After a line has been deleted by a *c* function, no further commands are attempted on the corpse.

If text is appended after a line by *a* or *r* functions, and the line is subsequently changed, the text inserted by the *c* function will be placed *before* the text of the *a* or *r* functions. (The *r* function is described in Section 3.4.)

Note: Within the text put in the output by these functions, leading blanks and tabs will disappear, as always in *sed* commands. To get leading blanks and tabs into the output, precede the first desired blank or tab by a backslash; the backslash will not appear in the output.

Example:

The list of editing commands:

```
n
a\
XXXX
d
```

applied to our standard input, produces:

```
In Xanadu did Kubhla Khan
XXXX
Where Alph, the sacred river, ran
XXXX
Down to a sunless sea.
```

In this particular case, the same effect would be produced by either of the two following command lists:

```
n          n
i\         c\
XXXX       XXXX
d
```

3.2. Substitute Function

One very important function changes parts of lines selected by a context search within the line.

(2)s<pattern><replacement><flags> -- substitute

The *s* function replaces *part* of a line (selected by <pattern>) with <replacement>. It can best be read:

Substitute for <pattern>, <replacement>

The <pattern> argument contains a pattern, exactly like the patterns in

addresses (see 2.2 above). The only difference between <pattern> and a context address is that the context address must be delimited by slash ('/') characters; <pattern> may be delimited by any character other than space or newline.

By default, only the first string matched by <pattern> is replaced, but see the *g* flag below.

The <replacement> argument begins immediately after the second delimiting character of <pattern>, and must be followed immediately by another instance of the delimiting character. (Thus there are exactly *three* instances of the delimiting character.)

The <replacement> is not a pattern, and the characters which are special in patterns do not have special meaning in <replacement>. Instead, other characters are special:

& is replaced by the string matched by <pattern>

d (where *d* is a single digit) is replaced by the *d*th substring matched by parts of <pattern> enclosed in '\(' and '\)'. If nested substrings occur in <pattern>, the *d*th is determined by counting opening delimiters ('\(').

As in patterns, special characters may be made literal by preceding them with backslash ('\').

The <flags> argument may contain the following flags:

g -- substitute <replacement> for all (non-overlapping) instances of <pattern> in the line. After a successful substitution, the scan for the next instance of <pattern> begins just after the end of the inserted characters; characters put into the line from <replacement> are not rescanned.

p -- print the line if a successful replacement was done. The *p* flag causes the line to be written to the output if and only if a substitution was actually made by the *s* function. Notice that if several *s* functions, each followed by a *p* flag, successfully substitute in the same input line, multiple copies of the line will be written to the output: one for each successful substitution.

w <filename> -- write the line to a file if a successful replacement was done. The *w* flag causes lines which are actually substituted by the *s* function to be written to a file named by <filename>. If <filename> exists before *sed* is run, it is overwritten; if not, it is created.

A single space must separate *w* and <filename>.

The possibilities of multiple, somewhat different copies of one input line being written are the same as for *p*.

A maximum of 10 different file names may be mentioned after *w* flags and *w* functions (see below), combined.

Examples:

The following command, applied to our standard input,

```
s/to/by/w changes
```

produces, on the standard output:

In Xanadu did Kubhla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless by man
Down by a sunless sea.

and, on the file 'changes':

Through caverns measureless by man
Down by a sunless sea.

If the nocopy option is in effect, the command:

s/[.,;?:]/*P&*/gp

produces:

A stately pleasure dome decree*P:*
Where Alph*P,* the sacred river*P,* ran
Down to a sunless sea*P.*

Finally, to illustrate the effect of the *g* flag, the command:

/X/s/an/AN/p

produces (assuming nocopy mode):

In XANadu did Kubhla Khan

and the command:

/X/s/an/AN/gp

produces:

In XANadu did Kubhla KhAN

3.3. Input-output Functions

(2)p -- print

The print function writes the addressed lines to the standard output file. They are written at the time the *p* function is encountered, regardless of what succeeding editing commands may do to the lines.

(2)w <filename> -- write on <filename>

The write function writes the addressed lines to the file named by <filename>. If the file previously existed, it is overwritten; if not, it is created. The lines are written exactly as they exist when the write function is encountered for each line, regardless of what subsequent editing commands may do to them.

Exactly one space must separate the *w* and <filename>.

A maximum of ten different files may be mentioned in write functions and *w* flags after *s* functions, combined.

(1)r <filename> -- read the contents of a file

The read function reads the contents of <filename>, and appends them after the line matched by the address. The file is read and appended regardless of what subsequent editing commands do to the line which matched its address. If *r* and *a* functions are executed on the same line, the text from the *a* functions and the *r* functions is written to the output in the order that the functions are executed.

Exactly one space must separate the *r* and <filename>. If a file mentioned by a *r* function cannot be opened, it is considered a null file, not an error, and no diagnostic is given.

NOTE: Since there is a limit to the number of files that can be opened simultaneously, care should be taken that no more than ten files be mentioned in *w* functions or flags; that number is reduced by one if any *r* functions are present. (Only one read file is open at one time.)

Examples

Assume that the file 'note1' has the following contents:

> Note: Kubla Khan (more properly Kublai Khan; 1216-1294) was the grandson and most eminent successor of Genghiz (Chingiz) Khan, and founder of the Mongol dynasty in China.

Then the following command:

```
/Kubla/r note1
```

produces:

```
In Xanadu did Kubla Khan
    Note: Kubla Khan (more properly Kublai Khan; 1216-1294) was the grandson
    and most eminent successor of Genghiz (Chingiz) Khan, and founder of the
    Mongol dynasty in China.
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

3.4. Multiple Input-line Functions

Three functions, all spelled with capital letters, deal specially with pattern spaces containing imbedded newlines; they are intended principally to provide pattern matches across lines in the input.

(2)N -- Next line

> The next input line is appended to the current line in the pattern space; the two input lines are separated by an imbedded newline. Pattern matches may extend across the imbedded newline(s).

(2)D -- Delete first part of the pattern space

> Delete up to and including the first newline character in the current pattern space. If the pattern space becomes empty (the only newline was the terminal newline), read another line from the input. In any case, begin the list of editing commands again from its beginning.

(2)P -- Print first part of the pattern space

> Print up to and including the first newline in the pattern space.

The *P* and *D* functions are equivalent to their lower-case counterparts if there are no imbedded newlines in the pattern space.

3.5. Hold and Get Functions

Four functions save and retrieve part of the input for possible later use.

(2)h -- hold pattern space

> The *h* functions copies the contents of the pattern space into a hold area (destroying the previous contents of the hold area).

(2)H -- Hold pattern space

> The *H* function appends the contents of the pattern space to the contents of the

hold area; the former and new contents are separated by a newline.

(2)g -- get contents of hold area

The *g* function copies the contents of the hold area into the pattern space (destroying the previous contents of the pattern space).

(2)G -- Get contents of hold area

The *G* function appends the contents of the hold area to the contents of the pattern space; the former and new contents are separated by a newline.

(2)x -- exchange

The exchange command interchanges the contents of the pattern space and the hold area.

Example

The commands

```
1h
1s/ did.*//
1x
G
s/\n/  :/
```

applied to our standard example, produce:

```
In Xanadu did Kubla Khan  :In Xanadu
A stately pleasure dome decree:  :In Xanadu
Where Alph, the sacred river, ran  :In Xanadu
Through caverns measureless to man  :In Xanadu
Down to a sunless sea.  :In Xanadu
```

3.6. Flow-of-Control Functions

These functions do no editing on the input lines, but control the application of functions to the lines selected by the address part.

(2)! -- Don't

The *Don't* command causes the next command (written on the same line), to be applied to all and only those input lines *not* selected by the adress part.

(2){ -- Grouping

The grouping command '{' causes the next set of commands to be applied (or not applied) as a block to the input lines selected by the addresses of the grouping command. The first of the commands under control of the grouping may appear on the same line as the '{' or on the next line.

The group of commands is terminated by a matching '}' standing on a line by itself.

Groups can be nested.

(0):<label> -- place a label

The label function marks a place in the list of editing commands which may be referred to by *b* and *t* functions. The <label> may be any sequence of eight or fewer characters; if two different colon functions have identical labels, a compile time diagnostic will be generated, and no execution attempted.

(2)b<label> -- branch to label

The branch function causes the sequence of editing commands being applied to the current input line to be restarted immediately after the place where a colon function with the same <label> was encountered. If no colon function with the same label can be found after all the editing commands have been compiled, a compile time diagnostic is produced, and no execution is attempted.

A *b* function with no <label> is taken to be a branch to the end of the list of editing commands; whatever should be done with the current input line is done, and another input line is read; the list of editing commands is restarted from the beginning on the new line.

(2)t<label> -- test substitutions

The *t* function tests whether *any* successful substitutions have been made on the current input line; if so, it branches to <label>; if not, it does nothing. The flag which indicates that a successful substitution has been executed is reset by:

1) reading a new input line, or
2) executing a *t* function.

3.7. Miscellaneous Functions

(1)= -- equals

The = function writes to the standard output the line number of the line matched by its address.

(1)q -- quit

The *q* function causes the current line to be written to the output (if it should be), any appended or read text to be written, and execution to be terminated.

Reference

[1] Ken Thompson and Dennis M. Ritchie, *The UNIX Programmer's Manual.* Bell Laboratories, 1978.

Awk — A Pattern Scanning and Processing Language (Second Edition)

Alfred V. Aho

Brian W. Kernighan

Peter J. Weinberger

ABSTRACT

Awk is a programming language whose basic operation is to search a set of files for patterns, and to perform specified actions upon lines or fields of lines which contain instances of those patterns. *Awk* makes certain data selection and transformation operations easy to express; for example, the *awk* program

length > 72

prints all input lines whose length exceeds 72 characters; the program

NF %2 == 0

prints all lines with an even number of fields; and the program

{ $1 = log($1); print }

replaces the first field of each line by its logarithm.

Awk patterns may include arbitrary boolean combinations of regular expressions and of relational operators on strings, numbers, fields, variables, and array elements. Actions may include the same pattern-matching constructions as in patterns, as well as arithmetic and string expressions and assignments, **if-else**, **while**, **for** statements, and multiple output streams.

This report contains a user's guide, a discussion of the design and implementation of *awk*, and some timing statistics.

September 1, 1978

Awk — A Pattern Scanning and Processing Language (Second Edition)

Alfred V. Aho

Brian W. Kernighan

Peter J. Weinberger

1. Introduction

Awk is a programming language designed to make many common information retrieval and text manipulation tasks easy to state and to perform.

The basic operation of *awk* is to scan a set of input lines in order, searching for lines which match any of a set of patterns which the user has specified. For each pattern, an action can be specified; this action will be performed on each line that matches the pattern.

Readers familiar with the UNIX† program *grep*[1] will recognize the approach, although in *awk* the patterns may be more general than in *grep*, and the actions allowed are more involved than merely printing the matching line. For example, the *awk* program

```
{print $3, $2}
```

prints the third and second columns of a table in that order. The program

```
$2 ~ /A|B|C/
```

prints all input lines with an A, B, or C in the second field. The program

```
$1 != prev  { print; prev = $1 }
```

prints all lines in which the first field is different from the previous first field.

1.1. Usage

The command

```
awk program [files]
```

executes the *awk* commands in the string **program** on the set of named files, or on the standard input if there are no files. The statements can also be placed in a file **pfile**, and executed by the command

```
awk -f pfile [files]
```

† UNIX is a trademark of Bell Laboratories.

1.2. Program Structure

An *awk* program is a sequence of statements of the form:

```
pattern   { action }
pattern   { action }
...
```

Each line of input is matched against each of the patterns in turn. For each pattern that matches, the associated action is executed. When all the patterns have been tested, the next line is fetched and the matching starts over.

Either the pattern or the action may be left out, but not both. If there is no action for a pattern, the matching line is simply copied to the output. (Thus a line which matches several patterns can be printed several times.) If there is no pattern for an action, then the action is performed for every input line. A line which matches no pattern is ignored.

Since patterns and actions are both optional, actions must be enclosed in braces to distinguish them from patterns.

1.3. Records and Fields

Awk input is divided into "records" terminated by a record separator. The default record separator is a newline, so by default *awk* processes its input a line at a time. The number of the current record is available in a variable named **NR**.

Each input record is considered to be divided into "fields." Fields are normally separated by white space — blanks or tabs — but the input field separator may be changed, as described below. Fields are referred to as **$1, $2,** and so forth, where **$1** is the first field, and **$0** is the whole input record itself. Fields may be assigned to. The number of fields in the current record is available in a variable named **NF**.

The variables **FS** and **RS** refer to the input field and record separators; they may be changed

at any time to any single character. The optional command-line argument **–F***c* may also be used to set **FS** to the character *c*.

If the record separator is empty, an empty input line is taken as the record separator, and blanks, tabs and newlines are treated as field separators.

The variable **FILENAME** contains the name of the current input file.

1.4. Printing

An action may have no pattern, in which case the action is executed for all lines. The simplest action is to print some or all of a record; this is accomplished by the *awk* command **print**. The *awk* program

```
{ print }
```

prints each record, thus copying the input to the output intact. More useful is to print a field or fields from each record. For instance,

```
print $2, $1
```

prints the first two fields in reverse order. Items separated by a comma in the print statement will be separated by the current output field separator when output. Items not separated by commas will be concatenated, so

```
print $1 $2
```

runs the first and second fields together.

The predefined variables **NF** and **NR** can be used; for example

```
{ print NR, NF, $0 }
```

prints each record preceded by the record number and the number of fields.

Output may be diverted to multiple files; the program

```
{ print $1 >"foo1"; print $2 >"foo2" }
```

writes the first field, **$1**, on the file **foo1**, and the second field on file **foo2**. The **>>** notation can also be used:

```
print $1 >>"foo"
```

appends the output to the file **foo**. (In each case, the output files are created if necessary.) The file name can be a variable or a field as well as a constant; for example,

```
print $1 >$2
```

uses the contents of field 2 as a file name.

Naturally there is a limit on the number of output files; currently it is 10.

Similarly, output can be piped into another process (on UNIX only); for instance,

```
print | "mail bwk"
```

mails the output to **bwk**.

The variables **OFS** and **ORS** may be used to change the current output field separator and output record separator. The output record separator is appended to the output of the **print** statement.

Awk also provides the **printf** statement for output formatting:

```
printf format expr, expr, ...
```

formats the expressions in the list according to the specification in **format** and prints them. For example,

```
printf "%8.2f  %10ld\n", $1, $2
```

prints **$1** as a floating point number 8 digits wide, with two after the decimal point, and **$2** as a 10-digit long decimal number, followed by a newline. No output separators are produced automatically; you must add them yourself, as in this example. The version of **printf** is identical to that used with C.[2]

2. Patterns

A pattern in front of an action acts as a selector that determines whether the action is to be executed. A variety of expressions may be used as patterns: regular expressions, arithmetic relational expressions, string-valued expressions, and arbitrary boolean combinations of these.

2.1. BEGIN and END

The special pattern **BEGIN** matches the beginning of the input, before the first record is read. The pattern **END** matches the end of the input, after the last record has been processed. **BEGIN** and **END** thus provide a way to gain control before and after processing, for initialization and wrapup.

As an example, the field separator can be set to a colon by

```
BEGIN   { FS = ":" }
... rest of program ...
```

Or the input lines may be counted by

```
END  { print NR }
```

If **BEGIN** is present, it must be the first pattern; **END** must be the last if used.

2.2. Regular Expressions

The simplest regular expression is a literal string of characters enclosed in slashes, like

```
/smith/
```

This is actually a complete *awk* program which will print all lines which contain any occurrence of the name "smith". If a line contains "smith" as part of a larger word, it will also be printed, as in

```
blacksmithing
```

Awk regular expressions include the regular expression forms found in the UNIX text editor *ed*[1] and *grep* (without back-referencing). In addition, *awk* allows parentheses for grouping, | for alternatives, + for "one or more", and ? for "zero or one", all as in *lex*. Character classes may be abbreviated: [a-zA-Z0-9] is the set of all letters and digits. As an example, the *awk* program

```
/[Aa]ho |[Ww]einberger |[Kk]ernighan/
```

will print all lines which contain any of the names "Aho," "Weinberger" or "Kernighan," whether capitalized or not.

Regular expressions (with the extensions listed above) must be enclosed in slashes, just as in *ed* and *sed*. Within a regular expression, blanks and the regular expression metacharacters are significant. To turn of the magic meaning of one of the regular expression characters, precede it with a backslash. An example is the pattern

```
/\/.*\//
```

which matches any string of characters enclosed in slashes.

One can also specify that any field or variable matches a regular expression (or does not match it) with the operators ~ and !~. The program

```
$1 ~ /[jJ]ohn/
```

prints all lines where the first field matches "john" or "John." Notice that this will also match "Johnson", "St. Johnsbury", and so on. To restrict it to exactly [jJ]ohn, use

```
$1 ~ /^[jJ]ohn$/
```

The caret ^ refers to the beginning of a line or field; the dollar sign $ refers to the end.

2.3. Relational Expressions

An *awk* pattern can be a relational expression involving the usual relational operators <, <=, ==, !=, >=, and >. An example is

```
$2 > $1 + 100
```

which selects lines where the second field is at least 100 greater than the first field. Similarly,

```
NF %2 == 0
```

prints lines with an even number of fields.

In relational tests, if neither operand is numeric, a string comparison is made; otherwise it is numeric. Thus,

```
$1 >= "s"
```

selects lines that begin with an s, t, u, etc. In the absence of any other information, fields are treated as strings, so the program

```
$1 > $2
```

will perform a string comparison.

2.4. Combinations of Patterns

A pattern can be any boolean combination of patterns, using the operators || (or), && (and), and ! (not). For example,

```
$1 >= "s" && $1 < "t" && $1 != "smith"
```

selects lines where the first field begins with "s", but is not "smith". && and || guarantee that their operands will be evaluated from left to right; evaluation stops as soon as the truth or falsehood is determined.

2.5. Pattern Ranges

The "pattern" that selects an action may also consist of two patterns separated by a comma, as in

```
pat1, pat2      { ... }
```

In this case, the action is performed for each line between an occurrence of **pat1** and the next occurrence of **pat2** (inclusive). For example,

```
/start/, /stop/
```

prints all lines between **start** and **stop**, while

```
NR == 100, NR == 200 { ... }
```

does the action for lines 100 through 200 of the input.

3. Actions

An *awk* action is a sequence of action statements terminated by newlines or semicolons. These action statements can be used to do a variety of bookkeeping and string manipulating tasks.

3.1. Built-in Functions

Awk provides a "length" function to compute the length of a string of characters. This program prints each record, preceded by its length:

```
{print length, $0}
```

length by itself is a "pseudo-variable" which yields the length of the current record; **length(argument)** is a function which yields the length of its argument, as in the equivalent

```
{print length($0), $0}
```

The argument may be any expression.

Awk also provides the arithmetic functions **sqrt**, **log**, **exp**, and **int**, for square root, base *e* logarithm, exponential, and integer part of their respective arguments.

The name of one of these built-in functions, without argument or parentheses, stands for the value of the function on the whole record. The program

```
length < 10 || length > 20
```

prints lines whose length is less than 10 or greater than 20.

The function **substr(s, m, n)** produces the substring of **s** that begins at position **m** (origin 1) and is at most **n** characters long. If **n** is omitted, the substring goes to the end of **s**. The function **index(s1, s2)** returns the position where the string **s2** occurs in **s1**, or zero if it does not.

The function **sprintf(f, e1, e2, ...)** produces the value of the expressions **e1**, **e2**, etc., in the **printf** format specified by **f**. Thus, for example,

```
x = sprintf("%8.2f %10d", $1, $2)
```

sets **x** to the string produced by formatting the values of **$1** and **$2**.

3.2. Variables, Expressions, and Assignments

Awk variables take on numeric (floating point) or string values according to context. For example, in

```
x = 1
```

x is clearly a number, while in

```
x = "smith"
```

it is clearly a string. Strings are converted to numbers and vice versa whenever context demands it. For instance,

```
x = "3" + "4"
```

assigns 7 to **x**. Strings which cannot be interpreted as numbers in a numerical context will generally have numeric value zero, but it is unwise to count on this behavior.

By default, variables (other than built-ins) are initialized to the null string, which has numerical value zero; this eliminates the need for most **BEGIN** sections. For example, the sums of the first two fields can be computed by

```
        { s1 += $1; s2 += $2 }
END     { print s1, s2 }
```

Arithmetic is done internally in floating point. The arithmetic operators are **+**, **-**, *****, **/**, and **%** (mod). The C increment **++** and decrement **--** operators are also available, and so are the assignment operators **+=**, **-=**, ***=**, **/=**, and **%=**. These operators may all be used in expressions.

3.3. Field Variables

Fields in *awk* share essentially all of the properties of variables — they may be used in arithmetic or string operations, and may be assigned to. Thus one can replace the first field with a sequence number like this:

```
{ $1 = NR; print }
```

or accumulate two fields into a third, like this:

```
{ $1 = $2 + $3; print $0 }
```

or assign a string to a field:

```
{ if ($3 > 1000)
      $3 = "too big"
  print
}
```

which replaces the third field by "too big" when it is, and in any case prints the record.

Field references may be numerical expressions, as in

```
{ print $i, $(i+1), $(i+n) }
```

Whether a field is deemed numeric or string depends on context; in ambiguous cases like

```
if ($1 == $2) ...
```

fields are treated as strings.

Each input line is split into fields automatically as necessary. It is also possible to split any variable or string into fields:

```
n = split(s, array, sep)
```

splits the the string **s** into **array[1]**, ..., **array[n]**. The number of elements found is returned. If the **sep** argument is provided, it is used as the field separator; otherwise **FS** is used as the separator.

3.4. String Concatenation

Strings may be concatenated. For example

```
length($1 $2 $3)
```

returns the length of the first three fields. Or in a **print** statement,

```
print $1 " is " $2
```

prints the two fields separated by " is ". Variables and numeric expressions may also appear in concatenations.

3.5. Arrays

Array elements are not declared; they spring into existence by being mentioned. Subscripts may have *any* non-null value, including non-numeric strings. As an example of a conventional numeric subscript, the statement

```
x[NR] = $0
```

assigns the current input record to the NR-th element of the array x. In fact, it is possible in principle (though perhaps slow) to process the entire input in a random order with the *awk* program

```
        { x[NR] = $0 }
END { ... program ... }
```

The first action merely records each input line in the array x.

Array elements may be named by non-numeric values, which gives *awk* a capability rather like the associative memory of Snobol tables. Suppose the input contains fields with values like **apple**, **orange**, etc. Then the program

```
/apple/   { x["apple"]++ }
/orange/  { x["orange"]++ }
END       { print x["apple"], x["orange"] }
```

increments counts for the named array elements, and prints them at the end of the input.

3.6. Flow-of-Control Statements

Awk provides the basic flow-of-control statements **if-else**, **while**, **for**, and statement grouping with braces, as in C. We showed the **if** statement in section 3.3 without describing it. The condition in parentheses is evaluated; if it is true, the statement following the **if** is done. The **else** part is optional.

The **while** statement is exactly like that of C. For example, to print all input fields one per line,

```
i = 1
while (i <= NF) {
    print $i
    ++i
}
```

The **for** statement is also exactly that of C:

```
for (i = 1; i <= NF; i++)
    print $i
```

does the same job as the **while** statement above.

There is an alternate form of the **for** statement which is suited for accessing the elements of an associative array:

```
for (i in array)
    statement
```

does *statement* with **i** set in turn to each element of **array**. The elements are accessed in an apparently random order. Chaos will ensue if **i** is altered, or if any new elements are accessed during the loop.

The expression in the condition part of an **if**, **while** or **for** can include relational operators like <, <=, >, >=, == ("is equal to"), and != ("not equal to"); regular expression matches with the match operators ~ and !~; the logical operators ||, &&, and !; and of course parentheses for grouping.

The **break** statement causes an immediate exit from an enclosing **while** or **for**; the **continue** statement causes the next iteration to begin.

The statement **next** causes *awk* to skip immediately to the next record and begin scanning the patterns from the top. The statement **exit** causes the program to behave as if the end of the input had occurred.

Comments may be placed in *awk* programs: they begin with the character # and end with the end of the line, as in

```
print x, y # this is a comment
```

4. Design

The UNIX system already provides several programs that operate by passing input through a selection mechanism. *Grep*, the first and simplest, merely prints all lines which match a single specified pattern. *Egrep* provides more general patterns, i.e., regular expressions in full generality; *fgrep* searches for a set of keywords with a particularly fast algorithm. *Sed*[1] provides most of the editing facilities of the editor *ed*, applied to a stream of input. None of these programs provides numeric capabilities, logical relations, or variables.

Lex[3] provides general regular expression recognition capabilities, and, by serving as a C program generator, is essentially open-ended in its capabilities. The use of *lex*, however, requires a knowledge of C programming, and a *lex* program must be compiled and loaded before use, which discourages its use for one-shot applications.

Awk is an attempt to fill in another part of the matrix of possibilities. It provides general regular expression capabilities and an implicit input/output loop. But it also provides convenient numeric processing, variables, more general selection, and control flow in the actions. It does

not require compilation or a knowledge of C. Finally, *awk* provides a convenient way to access fields within lines; it is unique in this respect.

Awk also tries to integrate strings and numbers completely, by treating all quantities as both string and numeric, deciding which representation is appropriate as late as possible. In most cases the user can simply ignore the differences.

Most of the effort in developing *awk* went into deciding what *awk* should or should not do (for instance, it doesn't do string substitution) and what the syntax should be (no explicit operator for concatenation) rather than on writing or debugging the code. We have tried to make the syntax powerful but easy to use and well adapted to scanning files. For example, the absence of declarations and implicit initializations, while probably a bad idea for a general-purpose programming language, is desirable in a language that is meant to be used for tiny programs that may even be composed on the command line.

In practice, *awk* usage seems to fall into two broad categories. One is what might be called "report generation" — processing an input to extract counts, sums, sub-totals, etc. This also includes the writing of trivial data validation programs, such as verifying that a field contains only numeric information or that certain delimiters are properly balanced. The combination of textual and numeric processing is invaluable here.

A second area of use is as a data transformer, converting data from the form produced by one program into that expected by another. The simplest examples merely select fields, perhaps with rearrangements.

5. Implementation

The actual implementation of *awk* uses the language development tools available on the UNIX operating system. The grammar is specified with *yacc*;[4] the lexical analysis is done by *lex*; the regular expression recognizers are deterministic finite automata constructed directly from the expressions. An *awk* program is translated into a parse tree which is then directly executed by a simple interpreter.

Awk was designed for ease of use rather than processing speed; the delayed evaluation of variable types and the necessity to break input into fields makes high speed difficult to achieve in any case. Nonetheless, the program has not proven to be unworkably slow.

Table I below shows the execution (user + system) time on a PDP-11/70 of the UNIX programs *wc*, *grep*, *egrep*, *fgrep*, *sed*, *lex*, and *awk* on the following simple tasks:

1. count the number of lines.
2. print all lines containing "doug".
3. print all lines containing "doug", "ken" or "dmr".
4. print the third field of each line.
5. print the third and second fields of each line, in that order.
6. append all lines containing "doug", "ken", and "dmr" to files "jdoug", "jken", and "jdmr", respectively.
7. print each line prefixed by "line-number : ".
8. sum the fourth column of a table.

The program *wc* merely counts words, lines and characters in its input; we have already mentioned the others. In all cases the input was a file containing 10,000 lines as created by the command *ls -l*; each line has the form

- rw- rw- rw- 1 ava 123 Oct 15 17:05 xxx

The total length of this input is 452,960 characters. Times for *lex* do not include compile or load.

As might be expected, *awk* is not as fast as the specialized tools *wc*, *sed*, or the programs in the *grep* family, but is faster than the more general tool *lex*. In all cases, the tasks were about as easy to express as *awk* programs as programs in these other languages; tasks involving fields were considerably easier to express as *awk* programs. Some of the test programs are shown in *awk*, *sed* and *lex*.

References

1. K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual,* Bell Laboratories, May 1975. Sixth Edition
2. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
3. M. E. Lesk, "Lex — A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey, October 1975.
4. S. C. Johnson, "Yacc — Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey, July 1975.

Program	Task 1	2	3	4	5	6	7	8
wc	8.6							
grep	11.7	13.1						
egrep	6.2	11.5	11.6					
fgrep	7.7	13.8	16.1					
sed	10.2	11.6	15.8	29.0	30.5	16.1		
lex	65.1	150.1	144.2	67.7	70.3	104.0	81.7	92.8
awk	15.0	25.6	29.9	33.3	38.9	46.4	71.4	31.1

Table I. Execution Times of Programs. (Times are in sec.)

The programs for some of these jobs are shown below. The *lex* programs are generally too long to show.

AWK:

1. `END {print NR}`

2. `/doug/`

3. `/ken|doug|dmr/`

4. `{print $3}`

5. `{print $3, $2}`

6.
```
/ken/     {print >"jken"}
/doug/    {print >"jdoug"}
/dmr/     {print >"jdmr"}
```

7. `{print NR ": " $0}`

8.
```
     {sum = sum + $4}
END {print sum}
```

SED:

1. `$=`

2. `/doug/p`

3.
```
/doug/p
/doug/d
/ken/p
/ken/d
/dmr/p
/dmr/d
```

4. `/[^ ]* [ ]*[^ ]* [ ]*\([^ ]*\) .*/s//\1/p`

5. `/[^ ]* [ ]*\([^ ]*\) [ ]*\([^ ]*\) .*/s//\2 \1/p`

6.
```
/ken/w jken
/doug/w jdoug
/dmr/w jdmr
```

LEX:

1.
```
%{
int i;
%}
%%
\n    i++;
.     ;
%%
yywrap() {
      printf("%d\n", i);
}
```

2.
```
%%
^.*doug.*$        printf("%s\n", yytext);
.     ;
\n    ;
```

DC – An Interactive Desk Calculator

Robert Morris

Lorinda Cherry

ABSTRACT

DC is an interactive desk calculator program implemented on the UNIX† time-sharing system to do arbitrary-precision integer arithmetic. It has provision for manipulating scaled fixed-point numbers and for input and output in bases other than decimal.

The size of numbers that can be manipulated is limited only by available core storage. On typical implementations of UNIX, the size of numbers that can be handled varies from several hundred digits on the smallest systems to several thousand on the largest.

September 12, 1986

† UNIX is a trademark of Bell Laboratories.

DC – An Interactive Desk Calculator

Robert Morris

Lorinda Cherry

DC is an arbitrary precision arithmetic package implemented on the UNIX time-sharing system in the form of an interactive desk calculator. It works like a stacking calculator using reverse Polish notation. Ordinarily DC operates on decimal integers, but one may specify an input base, output base, and a number of fractional digits to be maintained.

A language called BC [1] has been developed which accepts programs written in the familiar style of higher-level programming languages and compiles output which is interpreted by DC. Some of the commands described below were designed for the compiler interface and are not easy for a human user to manipulate.

Numbers that are typed into DC are put on a push-down stack. DC commands work by taking the top number or two off the stack, performing the desired operation, and pushing the result on the stack. If an argument is given, input is taken from that file until its end, then from the standard input.

SYNOPTIC DESCRIPTION

Here we describe the DC commands that are intended for use by people. The additional commands that are intended to be invoked by compiled output are described in the detailed description.

Any number of commands are permitted on a line. Blanks and new-line characters are ignored except within numbers and in places where a register name is expected.

The following constructions are recognized:

number

The value of the number is pushed onto the main stack. A number is an unbroken string of the digits 0-9 and the capital letters A–F which are treated as digits with values 10–15 respectively. The number may be preceded by an underscore to input a negative number. Numbers may contain decimal points.

+ - * % ^

The top two values on the stack are added (+), subtracted (–), multiplied (*), divided (/), remaindered (%), or exponentiated (^). The two entries are popped off the stack; the result is pushed on the stack in their place. The result of a division is an integer truncated toward zero. See the detailed description below for the treatment of numbers with decimal points. An exponent must not have any digits after the decimal point.

s*x*

The top of the main stack is popped and stored into a register named *x*, where *x* may be any character. If the **s** is capitalized, *x* is treated as a stack and the value is pushed onto it. Any character, even blank or new-line, is a valid register name.

l*x*

The value in register *x* is pushed onto the stack. The register *x* is not altered. If the **l** is capitalized, register *x* is treated as a stack and its top value is popped onto the main stack.

All registers start with empty value which is treated as a zero by the command **l** and is treated as an error by the command **L**.

d

The top value on the stack is duplicated.

p

The top value on the stack is printed. The top value remains unchanged.

f

All values on the stack and in registers are printed.

x

treats the top element of the stack as a character string, removes it from the stack, and executes it as a string of DC commands.

[... **]**

puts the bracketed character string onto the top of the stack.

q

exits the program. If executing a string, the recursion level is popped by two. If **q** is capitalized, the top value on the stack is popped and the string execution level is popped by that value.

<*x* **>***x* **=***x* **!<***x* **!>***x* **!=***x*

The top two elements of the stack are popped and compared. Register *x* is executed if they obey the stated relation. Exclamation point is negation.

v

replaces the top element on the stack by its square root. The square root of an integer is truncated to an integer. For the treatment of numbers with decimal points, see the detailed description below.

!

interprets the rest of the line as a UNIX command. Control returns to DC when the UNIX command terminates.

c

All values on the stack are popped; the stack becomes empty.

i

The top value on the stack is popped and used as the number radix for further input. If **i** is capitalized, the value of the input base is pushed onto the stack. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 or greater than 16.

o

The top value on the stack is popped and used as the number radix for further output. If **o** is capitalized, the value of the output base is pushed onto the stack.

k

The top of the stack is popped, and that value is used as a scale factor that influences the number of decimal places that are maintained during multiplication, division, and exponentiation. The scale factor must be greater than or equal to zero and less than 100. If **k** is capitalized, the value of the scale factor is pushed onto the stack.

z

The value of the stack level is pushed onto the stack.

?

A line of input is taken from the input source (usually the console) and executed.

DETAILED DESCRIPTION

Internal Representation of Numbers

Numbers are stored internally using a dynamic storage allocator. Numbers are kept in the form of a string of digits to the base 100 stored one digit per byte (centennial digits). The string is stored with the low-order digit at the beginning of the string. For example, the representation of 157 is 57,1. After any arithmetic operation on a number, care is taken that all digits are in the range 0–99 and that the number has no leading zeros. The number zero is represented by the empty string.

Negative numbers are represented in the 100's complement notation, which is analogous to two's complement notation for binary numbers. The high order digit of a negative number is always –1 and all other digits are in the range 0–99. The digit preceding the high order –1 digit is never a 99. The representation of –157 is 43,98,–1. We shall call this the canonical form of a number. The advantage of this kind of representation of negative numbers is ease of addition. When addition is performed digit by digit, the result is formally correct. The result need only be modified, if necessary, to put it into canonical form.

Because the largest valid digit is 99 and the byte can hold numbers twice that large, addition can be carried out and the handling of carries done later when that is convenient, as it sometimes is.

An additional byte is stored with each number beyond the high order digit to indicate the number of assumed decimal digits after the decimal point. The representation of .001 is 1,*3* where the scale has been italicized to emphasize the fact that it is not the high order digit. The value of this extra byte is called the **scale factor** of the number.

The Allocator

DC uses a dynamic string storage allocator for all of its internal storage. All reading and writing of numbers internally is done through the allocator. Associated with each string in the allocator is a four-word header containing pointers to the beginning of the string, the end of the string, the next place to write, and the next place to read. Communication between the allocator and DC is done via pointers to these headers.

The allocator initially has one large string on a list of free strings. All headers except the one pointing to this string are on a list of free headers. Requests for strings are made by size. The size of the string actually supplied is the next higher power of 2. When a request for a string is made, the allocator first checks the free list to see if there is a string of the desired size. If none is found, the allocator finds the next larger free string and splits it repeatedly until it has a string of the right size. Left-over strings are put on the free list. If there are no larger strings, the allocator

tries to coalesce smaller free strings into larger ones. Since all strings are the result of splitting large strings, each string has a neighbor that is next to it in core and, if free, can be combined with it to make a string twice as long. This is an implementation of the 'buddy system' of allocation described in [2].

Failing to find a string of the proper length after coalescing, the allocator asks the system for more space. The amount of space on the system is the only limitation on the size and number of strings in DC. If at any time in the process of trying to allocate a string, the allocator runs out of headers, it also asks the system for more space.

There are routines in the allocator for reading, writing, copying, rewinding, forward-spacing, and backspacing strings. All string manipulation is done using these routines.

The reading and writing routines increment the read pointer or write pointer so that the characters of a string are read or written in succession by a series of read or write calls. The write pointer is interpreted as the end of the information-containing portion of a string and a call to read beyond that point returns an end-of-string indication. An attempt to write beyond the end of a string causes the allocator to allocate a larger space and then copy the old string into the larger block.

Internal Arithmetic

All arithmetic operations are done on integers. The operands (or operand) needed for the operation are popped from the main stack and their scale factors stripped off. Zeros are added or digits removed as necessary to get a properly scaled result from the internal arithmetic routine. For example, if the scale of the operands is different and decimal alignment is required, as it is for addition, zeros are appended to the operand with the smaller scale. After performing the required arithmetic operation, the proper scale factor is appended to the end of the number before it is pushed on the stack.

A register called **scale** plays a part in the results of most arithmetic operations. **scale** is the bound on the number of decimal places retained in arithmetic computations. **scale** may be set to the number on the top of the stack truncated to an integer with the **k** command. **K** may be used to push the value of **scale** on the stack. **scale** must be greater than or equal to 0 and less than 100. The descriptions of the individual arithmetic operations will include the exact effect of **scale** on the computations.

Addition and Subtraction

The scales of the two numbers are compared and trailing zeros are supplied to the number with the lower scale to give both numbers the same scale. The number with the smaller scale is multiplied by 10 if the difference of the scales is odd. The scale of the result is then set to the larger of the scales of the two operands.

Subtraction is performed by negating the number to be subtracted and proceeding as in addition.

Finally, the addition is performed digit by digit from the low order end of the number. The carries are propagated in the usual way. The resulting number is brought into canonical form, which may require stripping of leading zeros, or for negative numbers replacing the high-order configuration 99,–1 by the digit –1. In any case, digits which are not in the range 0–99 must be brought into that range, propagating any carries or borrows that result.

Multiplication

The scales are removed from the two operands and saved. The operands are both made positive. Then multiplication is performed in a digit by digit manner that exactly mimics the hand method of multiplying. The first number is multiplied by each digit of the second number, beginning with its low order digit. The intermediate products are accumulated into a partial sum which becomes the final product. The product is put into the canonical form and its sign is computed from the signs of the original operands.

The scale of the result is set equal to the sum of the scales of the two operands. If that scale is larger than the internal register **scale** and also larger than both of the scales of the two operands, then the scale of the result is set equal to the largest of these three last quantities.

Division

The scales are removed from the two operands. Zeros are appended or digits removed from the dividend to make the scale of the result of the integer division equal to the internal quantity **scale**. The signs are removed and saved.

Division is performed much as it would be done by hand. The difference of the lengths of the two numbers is computed. If the divisor is longer than the dividend, zero is returned. Otherwise the top digit of the divisor is divided into the top two digits of the dividend. The result is used as the first (high-order) digit of the quotient. It may turn out be one unit too low, but if it is, the next trial quotient will be larger than 99 and this will be adjusted at the end of the process. The trial digit is multiplied by the divisor and the result subtracted from the dividend and the process is repeated to get additional quotient digits until the remaining dividend is smaller than the divisor. At the end, the digits of the quotient are put into the canonical form, with propagation of carry as needed. The sign is set from the sign of the operands.

Remainder

The division routine is called and division is performed exactly as described. The quantity returned is the remains of the dividend at the end of the divide process. Since division truncates toward zero, remainders have the same sign as the dividend. The scale of the remainder is set to the maximum of the scale of the dividend and the scale of the quotient plus the scale of the divisor.

Square Root

The scale is stripped from the operand. Zeros are added if necessary to make the integer result have a scale that is the larger of the internal quantity **scale** and the scale of the operand.

The method used to compute sqrt(y) is Newton's method with successive approximations by the rule

$$x_{n+1} = ½(x_n + \frac{y}{x_n})$$

The initial guess is found by taking the integer square root of the top two digits.

Exponentiation

Only exponents with zero scale factor are handled. If the exponent is zero, then the result is 1. If the exponent is negative, then it is made positive and the base is divided into one. The scale of the base is removed.

The integer exponent is viewed as a binary number. The base is repeatedly squared and the result is obtained as a product of those powers of the base that correspond to the positions of the one-bits in the binary representation of the exponent. Enough digits of the result are removed to make the scale of the result the same as if the indicated multiplication had been performed.

Input Conversion and Base

Numbers are converted to the internal representation as they are read in. The scale stored with a number is simply the number of fractional digits input. Negative numbers are indicated by preceding the number with a _. The hexadecimal digits A–F correspond to the numbers 10–15 regardless of input base. The **i** command can be used to change the base of the input numbers. This command pops the stack, truncates the resulting number to an integer, and uses it as the input base for all further input. The input base is initialized to 10 but may, for example be changed to 8 or 16 to do octal or hexadecimal to decimal conversions. The command **I** will push

the value of the input base on the stack.

Output Commands

The command **p** causes the top of the stack to be printed. It does not remove the top of the stack. All of the stack and internal registers can be output by typing the command **f**. The **o** command can be used to change the output base. This command uses the top of the stack, truncated to an integer as the base for all further output. The output base in initialized to 10. It will work correctly for any base. The command **O** pushes the value of the output base on the stack.

Output Format and Base

The input and output bases only affect the interpretation of numbers on input and output; they have no effect on arithmetic computations. Large numbers are output with 70 characters per line; a \ indicates a continued line. All choices of input and output bases work correctly, although not all are useful. A particularly useful output base is 100000, which has the effect of grouping digits in fives. Bases of 8 and 16 can be used for decimal-octal or decimal-hexadecimal conversions.

Internal Registers

Numbers or strings may be stored in internal registers or loaded on the stack from registers with the commands **s** and **l**. The command **s***x* pops the top of the stack and stores the result in register **x**. *x* can be any character. **l***x* puts the contents of register **x** on the top of the stack. The **l** command has no effect on the contents of register *x*. The **s** command, however, is destructive.

Stack Commands

The command **c** clears the stack. The command **d** pushes a duplicate of the number on the top of the stack on the stack. The command **z** pushes the stack size on the stack. The command **X** replaces the number on the top of the stack with its scale factor. The command **Z** replaces the top of the stack with its length.

Subroutine Definitions and Calls

Enclosing a string in [] pushes the ascii string on the stack. The **q** command quits or in executing a string, pops the recursion levels by two.

Internal Registers – Programming DC

The load and store commands together with [] to store strings, **x** to execute and the testing commands '<', '>', '=', '!<', '!>', '!=' can be used to program DC. The **x** command assumes the top of the stack is an string of DC commands and executes it. The testing commands compare the top two elements on the stack and if the relation holds, execute the register that follows the relation. For example, to print the numbers 0-9,

```
[lip1+ si li10>a]sa
0si lax
```

Push-Down Registers and Arrays

These commands were designed for used by a compiler, not by people. They involve push-down registers and arrays. In addition to the stack that commands work on, DC can be thought of as having individual stacks for each register. These registers are operated on by the commands **S** and **L**. **S***x* pushes the top value of the main stack onto the stack for the register *x*. **L***x* pops the stack for register *x* and puts the result on the main stack. The commands **s** and **l** also work on registers but not as push-down stacks. **l** doesn't effect the top of the register stack, and **s** destroys what was there before.

The commands to work on arrays are **:** and **;**. **:***x* pops the stack and uses this value as an index into the array *x*. The next element on the stack is stored at this index in *x*. An index must

be greater than or equal to 0 and less than 2048. ;*x* is the command to load the main stack from the array *x*. The value on the top of the stack is the index into the array *x* of the value to be loaded.

Miscellaneous Commands

The command ! interprets the rest of the line as a UNIX command and passes it to UNIX to execute. One other compiler command is **Q**. This command uses the top of the stack as the number of levels of recursion to skip.

DESIGN CHOICES

The real reason for the use of a dynamic storage allocator was that a general purpose program could be (and in fact has been) used for a variety of other tasks. The allocator has some value for input and for compiling (i.e. the bracket [...] commands) where it cannot be known in advance how long a string will be. The result was that at a modest cost in execution time, all considerations of string allocation and sizes of strings were removed from the remainder of the program and debugging was made easier. The allocation method used wastes approximately 25% of available space.

The choice of 100 as a base for internal arithmetic seemingly has no compelling advantage. Yet the base cannot exceed 127 because of hardware limitations and at the cost of 5% in space, debugging was made a great deal easier and decimal output was made much faster.

The reason for a stack-type arithmetic design was to permit all DC commands from addition to subroutine execution to be implemented in essentially the same way. The result was a considerable degree of logical separation of the final program into modules with very little communication between modules.

The rationale for the lack of interaction between the scale and the bases was to provide an understandable means of proceeding after a change of base or scale when numbers had already been entered. An earlier implementation which had global notions of scale and base did not work out well. If the value of **scale** were to be interpreted in the current input or output base, then a change of base or scale in the midst of a computation would cause great confusion in the interpretation of the results. The current scheme has the advantage that the value of the input and output bases are only used for input and output, respectively, and they are ignored in all other operations. The value of scale is not used for any essential purpose by any part of the program and it is used only to prevent the number of decimal places resulting from the arithmetic operations from growing beyond all bounds.

The design rationale for the choices for the scales of the results of arithmetic were that in no case should any significant digits be thrown away if, on appearances, the user actually wanted them. Thus, if the user wants to add the numbers 1.5 and 3.517, it seemed reasonable to give him the result 5.017 without requiring him to unnecessarily specify his rather obvious requirements for precision.

On the other hand, multiplication and exponentiation produce results with many more digits than their operands and it seemed reasonable to give as a minimum the number of decimal places in the operands but not to give more than that number of digits unless the user asked for them by specifying a value for **scale**. Square root can be handled in just the same way as multiplication. The operation of division gives arbitrarily many decimal places and there is simply no way to guess how many places the user wants. In this case only, the user must specify a **scale** to get any decimal places at all.

The scale of remainder was chosen to make it possible to recreate the dividend from the quotient and remainder. This is easy to implement; no digits are thrown away.

References

[1] L. L. Cherry, R. Morris, *BC - An Arbitrary Precision Desk-Calculator Language.*

[2] K. C. Knowlton, *A Fast Storage Allocator,* Comm. ACM **8**, pp. 623-625 (Oct. 1965).

BC – An Arbitrary Precision Desk-Calculator Language

Lorinda Cherry

Robert Morris

ABSTRACT

BC is a language and a compiler for doing arbitrary precision arithmetic on the PDP-11 under the UNIX† time-sharing system. The output of the compiler is interpreted and executed by a collection of routines which can input, output, and do arithmetic on indefinitely large integers and on scaled fixed-point numbers.

These routines are themselves based on a dynamic storage allocator. Overflow does not occur until all available core storage is exhausted.

The language has a complete control structure as well as immediate-mode operation. Functions can be defined and saved for later execution.

Two five hundred-digit numbers can be multiplied to give a thousand digit result in about ten seconds.

A small collection of library functions is also available, including sin, cos, arctan, log, exponential, and Bessel functions of integer order.

Some of the uses of this compiler are

- to do computation with large integers,
- to do computation accurate to many decimal places,
- conversion of numbers from one base to another base.

September 12, 1986

† UNIX is a trademark of Bell Laboratories.

BC – An Arbitrary Precision Desk-Calculator Language

Lorinda Cherry

Robert Morris

Introduction

BC is a language and a compiler for doing arbitrary precision arithmetic on the UNIX time-sharing system [1]. The compiler was written to make conveniently available a collection of routines (called DC [5]) which are capable of doing arithmetic on integers of arbitrary size. The compiler is by no means intended to provide a complete programming language. It is a minimal language facility.

There is a scaling provision that permits the use of decimal point notation. Provision is made for input and output in bases other than decimal. Numbers can be converted from decimal to octal by simply setting the output base to equal 8.

The actual limit on the number of digits that can be handled depends on the amount of storage available on the machine. Manipulation of numbers with many hundreds of digits is possible even on the smallest versions of UNIX.

The syntax of BC has been deliberately selected to agree substantially with the C language [2]. Those who are familiar with C will find few surprises in this language.

Simple Computations with Integers

The simplest kind of statement is an arithmetic expression on a line by itself. For instance, if you type in the line:

```
142857 + 285714
```

the program responds immediately with the line

```
428571
```

The operators –, *, /, %, and ^ can also be used; they indicate subtraction, multiplication, division, remaindering, and exponentiation, respectively. Division of integers produces an integer result truncated toward zero. Division by zero produces an error comment.

Any term in an expression may be prefixed by a minus sign to indicate that it is to be negated (the 'unary' minus sign). The expression

```
7+-3
```

is interpreted to mean that –3 is to be added to 7.

More complex expressions with several operators and with parentheses are interpreted just as in Fortran, with ^ having the greatest binding power, then * and % and /, and finally + and –. Contents of parentheses are evaluated before material outside the parentheses. Exponentiations are performed from right to left and the other operators from left to right. The two expressions

```
a^b^c  and  a^(b^c)
```

are equivalent, as are the two expressions

```
a*b*c  and  (a*b)*c
```

BC shares with Fortran and C the undesirable convention that

a/b*c is equivalent to (a/b)*c

Internal storage registers to hold numbers have single lower-case letter names. The value of an expression can be assigned to a register in the usual way. The statement

x = x + 3

has the effect of increasing by three the value of the contents of the register named x. When, as in this case, the outermost operator is an =, the assignment is performed but the result is not printed. Only 26 of these named storage registers are available.

There is a built-in square root function whose result is truncated to an integer (but see scaling below). The lines

```
x = sqrt(191)
x
```

produce the printed result

13

Bases

There are special internal quantities, called 'ibase' and 'obase'. The contents of 'ibase', initially set to 10, determines the base used for interpreting numbers read in. For example, the lines

```
ibase = 8
11
```

will produce the output line

9

and you are all set up to do octal to decimal conversions. Beware, however of trying to change the input base back to decimal by typing

ibase = 10

Because the number 10 is interpreted as octal, this statement will have no effect. For those who deal in hexadecimal notation, the characters A–F are permitted in numbers (no matter what base is in effect) and are interpreted as digits having values 10–15 respectively. The statement

ibase = A

will change you back to decimal input base no matter what the current input base is. Negative and large positive input bases are permitted but useless. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 and greater than 16.

The contents of 'obase', initially set to 10, are used as the base for output numbers. The lines

```
obase = 16
1000
```

will produce the output line

3E8

which is to be interpreted as a 3-digit hexadecimal number. Very large output bases are permitted, and they are sometimes useful. For example, large numbers can be output in groups of five digits by setting 'obase' to 100000. Strange (i.e. 1, 0, or negative) output bases are handled appropriately.

Very large numbers are split across lines with 70 characters per line. Lines which are continued end with \. Decimal output conversion is practically instantaneous, but output of very large

numbers (i.e., more than 100 digits) with other bases is rather slow. Non-decimal output conversion of a one hundred digit number takes about three seconds.

It is best to remember that 'ibase' and 'obase' have no effect whatever on the course of internal computation or on the evaluation of expressions, but only affect input and output conversion, respectively.

Scaling

A third special internal quantity called 'scale' is used to determine the scale of calculated quantities. Numbers may have up to 99 decimal digits after the decimal point. This fractional part is retained in further computations. We refer to the number of digits after the decimal point of a number as its scale.

When two scaled numbers are combined by means of one of the arithmetic operations, the result has a scale determined by the following rules. For addition and subtraction, the scale of the result is the larger of the scales of the two operands. In this case, there is never any truncation of the result. For multiplications, the scale of the result is never less than the maximum of the two scales of the operands, never more than the sum of the scales of the operands and, subject to those two restrictions, the scale of the result is set equal to the contents of the internal quantity 'scale'. The scale of a quotient is the contents of the internal quantity 'scale'. The scale of a remainder is the sum of the scales of the quotient and the divisor. The result of an exponentiation is scaled as if the implied multiplications were performed. An exponent must be an integer. The scale of a square root is set to the maximum of the scale of the argument and the contents of 'scale'.

All of the internal operations are actually carried out in terms of integers, with digits being discarded when necessary. In every case where digits are discarded, truncation and not rounding is performed.

The contents of 'scale' must be no greater than 99 and no less than 0. It is initially set to 0. In case you need more than 99 fraction digits, you may arrange your own scaling.

The internal quantities 'scale', 'ibase', and 'obase' can be used in expressions just like other variables. The line

```
scale = scale + 1
```

increases the value of 'scale' by one, and the line

```
scale
```

causes the current value of 'scale' to be printed.

The value of 'scale' retains its meaning as a number of decimal digits to be retained in internal computation even when 'ibase' or 'obase' are not equal to 10. The internal computations (which are still conducted in decimal, regardless of the bases) are performed to the specified number of decimal digits, never hexadecimal or octal or any other kind of digits.

Functions

The name of a function is a single lower-case letter. Function names are permitted to collide with simple variable names. Twenty-six different defined functions are permitted in addition to the twenty-six variable names. The line

```
define a(x){
```

begins the definition of a function with one argument. This line must be followed by one or more statements, which make up the body of the function, ending with a right brace }. Return of control from a function occurs when a return statement is executed or when the end of the function is reached. The return statement can take either of the two forms

```
return
return(x)
```

In the first case, the value of the function is 0, and in the second, the value of the expression in parentheses.

Variables used in the function can be declared as automatic by a statement of the form

```
auto x,y,z
```

There can be only one 'auto' statement in a function and it must be the first statement in the definition. These automatic variables are allocated space and initialized to zero on entry to the function and thrown away on return. The values of any variables with the same names outside the function are not disturbed. Functions may be called recursively and the automatic variables at each level of call are protected. The parameters named in a function definition are treated in the same way as the automatic variables of that function with the single exception that they are given a value on entry to the function. An example of a function definition is

```
define a(x,y){
        auto z
        z = x*y
        return(z)
}
```

The value of this function, when called, will be the product of its two arguments.

A function is called by the appearance of its name followed by a string of arguments enclosed in parentheses and separated by commas. The result is unpredictable if the wrong number of arguments is used.

Functions with no arguments are defined and called using parentheses with nothing between them: b().

If the function *a* above has been defined, then the line

```
a(7,3.14)
```

would cause the result 21.98 to be printed and the line

```
x = a(a(3,4),5)
```

would cause the value of x to become 60.

Subscripted Variables

A single lower-case letter variable name followed by an expression in brackets is called a subscripted variable (an array element). The variable name is called the array name and the expression in brackets is called the subscript. Only one-dimensional arrays are permitted. The names of arrays are permitted to collide with the names of simple variables and function names. Any fractional part of a subscript is discarded before use. Subscripts must be greater than or equal to zero and less than or equal to 2047.

Subscripted variables may be freely used in expressions, in function calls, and in return statements.

An array name may be used as an argument to a function, or may be declared as automatic in a function definition by the use of empty brackets:

```
f(a[])
define f(a[])
auto a[]
```

When an array name is so used, the whole contents of the array are copied for the use of the function, and thrown away on exit from the function. Array names which refer to whole arrays cannot be used in any other contexts.

Control Statements

The 'if', the 'while', and the 'for' statements may be used to alter the flow within programs or to cause iteration. The range of each of them is a statement or a compound statement consisting of a collection of statements enclosed in braces. They are written in the following way

```
if(relation) statement
while(relation) statement
for(expression1; relation; expression2) statement
```

or

```
if(relation) {statements}
while(relation) {statements}
for(expression1; relation; expression2) {statements}
```

A relation in one of the control statements is an expression of the form

```
x>y
```

where two expressions are related by one of the six relational operators <, >, <=, >=, ==, or !=. The relation == stands for 'equal to' and != stands for 'not equal to'. The meaning of the remaining relational operators is clear.

BEWARE of using = instead of == in a relational. Unfortunately, both of them are legal, so you will not get a diagnostic message, but = really will not do a comparison.

The 'if' statement causes execution of its range if and only if the relation is true. Then control passes to the next statement in sequence.

The 'while' statement causes execution of its range repeatedly as long as the relation is true. The relation is tested before each execution of its range and if the relation is false, control passes to the next statement beyond the range of the while.

The 'for' statement begins by executing 'expression1'. Then the relation is tested and, if true, the statements in the range of the 'for' are executed. Then 'expression2' is executed. The relation is tested, and so on. The typical use of the 'for' statement is for a controlled iteration, as in the statement

```
for(i=1; i<=10; i=i+1) i
```

which will print the integers from 1 to 10. Here are some examples of the use of the control statements.

```
define f(n){
auto i, x
x=1
for(i=1; i<=n; i=i+1) x=x*i
return(x)
}
```

The line

```
f(a)
```

will print *a* factorial if *a* is a positive integer. Here is the definition of a function which will compute values of the binomial coefficient (m and n are assumed to be positive integers).

```
define b(n,m){
auto x, j
x=1
for(j=1; j<=m; j=j+1) x=x*(n-j+1)/j
return(x)
}
```

The following function computes values of the exponential function by summing the appropriate series without regard for possible truncation errors:

```
scale = 20
define e(x){
        auto a, b, c, d, n
        a = 1
        b = 1
        c = 1
        d = 0
        n = 1
        while(1==1){
                a = a*x
                b = b*n
                c = c + a/b
                n = n + 1
                if(c==d) return(c)
                d = c
        }
}
```

Some Details

There are some language features that every user should know about even if he will not use them.

Normally statements are typed one to a line. It is also permissible to type several statements on a line separated by semicolons.

If an assignment statement is parenthesized, it then has a value and it can be used anywhere that an expression can. For example, the line

```
(x=y+17)
```

not only makes the indicated assignment, but also prints the resulting value.

Here is an example of a use of the value of an assignment statement even when it is not parenthesized.

```
x = a[i=i+1]
```

causes a value to be assigned to x and also increments i before it is used as a subscript.

The following constructs work in BC in exactly the same manner as they do in the C language. Consult the appendix or the C manuals [2] for their exact workings.

x=y=z is the same as	x=(y=z)
x =+ y	x = x+y
x =- y	x = x-y
x =* y	x = x*y
x =/ y	x = x/y
x =% y	x = x%y
x =^ y	x = x^y
x++	(x=x+1)-1
x—	(x=x-1)+1
++x	x = x+1
—x	x = x-1

Even if you don't intend to use the constructs, if you type one inadvertently, something correct but unexpected may happen.

WARNING! In some of these constructions, spaces are significant. There is a real difference between x=-y and x= -y. The first replaces x by x-y and the second by -y.

Three Important Things

1. To exit a BC program, type 'quit'.

2. There is a comment convention identical to that of C and of PL/I. Comments begin with '/*' and end with '*/'.

3. There is a library of math functions which may be obtained by typing at command level

   bc -l

This command will load a set of library functions which, at the time of writing, consists of sine (named 's'), cosine ('c'), arctangent ('a'), natural logarithm ('l'), exponential ('e') and Bessel functions of integer order ('j(n,x)'). Doubtless more functions will be added in time. The library sets the scale to 20. You can reset it to something else if you like. The design of these mathematical library routines is discussed elsewhere [3].

If you type

   bc file ...

BC will read and execute the named file or files before accepting commands from the keyboard. In this way, you may load your favorite programs and function definitions.

Acknowledgement

The compiler is written in YACC [4]; its original version was written by S. C. Johnson.

References

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

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

[3] R. Morris, *A Library of Reference Standard Mathematical Subroutines,* Bell Laboratories internal memorandum, 1975.

[4] S. C. Johnson, *YACC — Yet Another Compiler-Compiler.* Bell Laboratories Computing Science Technical Report #32, 1978.

[5] R. Morris and L. L. Cherry, *DC - An Interactive Desk Calculator.*

Appendix

1. Notation

In the following pages syntactic categories are in *italics*; literals are in **bold**; material in brackets [] is optional.

2. Tokens

Tokens consist of keywords, identifiers, constants, operators, and separators. Token separators may be blanks, tabs or comments. Newline characters or semicolons separate statements.

2.1. Comments

Comments are introduced by the characters /* and terminated by */.

2.2. Identifiers

There are three kinds of identifiers – ordinary identifiers, array identifiers and function identifiers. All three types consist of single lower-case letters. Array identifiers are followed by square brackets, possibly enclosing an expression describing a subscript. Arrays are singly dimensioned and may contain up to 2048 elements. Indexing begins at zero so an array may be indexed from 0 to 2047. Subscripts are truncated to integers. Function identifiers are followed by parentheses, possibly enclosing arguments. The three types of identifiers do not conflict; a program can have a variable named **x**, an array named **x** and a function named **x**, all of which are separate and distinct.

2.3. Keywords

The following are reserved keywords:

ibase	**if**
obase	**break**
scale	**define**
sqrt	**auto**
length	**return**
while	**quit**
for	

2.4. Constants

Constants consist of arbitrarily long numbers with an optional decimal point. The hexadecimal digits **A**–**F** are also recognized as digits with values 10–15, respectively.

3. Expressions

The value of an expression is printed unless the main operator is an assignment. Precedence is the same as the order of presentation here, with highest appearing first. Left or right associativity, where applicable, is discussed with each operator.

3.1. Primitive expressions

3.1.1. Named expressions

Named expressions are places where values are stored. Simply stated, named expressions are legal on the left side of an assignment. The value of a named expression is the value stored in the place named.

3.1.1.1. *identifiers*

Simple identifiers are named expressions. They have an initial value of zero.

3.1.1.2. *array-name* [*expression*]

Array elements are named expressions. They have an initial value of zero.

3.1.1.3. **scale**, **ibase** and **obase**

The internal registers **scale**, **ibase** and **obase** are all named expressions. **scale** is the number of digits after the decimal point to be retained in arithmetic operations. **scale** has an initial value of zero. **ibase** and **obase** are the input and output number radix respectively. Both **ibase** and **obase** have initial values of 10.

3.1.2. Function calls

3.1.2.1. *function-name* ([*expression* [, *expression* . . .]])

A function call consists of a function name followed by parentheses containing a comma-separated list of expressions, which are the function arguments. A whole array passed as an argument is specified by the array name followed by empty square brackets. All function arguments are passed by value. As a result, changes made to the formal parameters have no effect on the actual arguments. If the function terminates by executing a return statement, the value of the function is the value of the expression in the parentheses of the return statement or is zero if no expression is provided or if there is no return statement.

3.1.2.2. **sqrt** (*expression*)

The result is the square root of the expression. The result is truncated in the least significant decimal place. The scale of the result is the scale of the expression or the value of **scale,** whichever is larger.

3.1.2.3. **length** (*expression*)

The result is the total number of significant decimal digits in the expression. The scale of the result is zero.

3.1.2.4. **scale** (*expression*)

The result is the scale of the expression. The scale of the result is zero.

3.1.3. Constants

Constants are primitive expressions.

3.1.4. Parentheses

An expression surrounded by parentheses is a primitive expression. The parentheses are used to alter the normal precedence.

3.2. Unary operators

The unary operators bind right to left.

3.2.1. – *expression*

The result is the negative of the expression.

3.2.2. ++ *named-expression*

The named expression is incremented by one. The result is the value of the named expression after incrementing.

3.2.3. — *named-expression*

The named expression is decremented by one. The result is the value of the named expression after decrementing.

3.2.4. *named-expression* ++

The named expression is incremented by one. The result is the value of the named expression before incrementing.

3.2.5. *named-expression* —

The named expression is decremented by one. The result is the value of the named expression before decrementing.

3.3. Exponentiation operator

The exponentiation operator binds right to left.

3.3.1. *expression* ^ *expression*

The result is the first expression raised to the power of the second expression. The second expression must be an integer. If a is the scale of the left expression and b is the absolute value of the right expression, then the scale of the result is:

$\min(a \times b, \max(\textbf{scale}, a))$

3.4. Multiplicative operators

The operators *, /, % bind left to right.

3.4.1. *expression* * *expression*

The result is the product of the two expressions. If a and b are the scales of the two expressions, then the scale of the result is:

$\min(a+b, \max(\textbf{scale}, a, b))$

3.4.2. *expression* / *expression*

The result is the quotient of the two expressions. The scale of the result is the value of **scale**.

3.4.3. *expression* % *expression*

The % operator produces the remainder of the division of the two expressions. More precisely, $a\%b$ is $a-a/b*b$.

The scale of the result is the sum of the scale of the divisor and the value of **scale**

3.5. Additive operators

The additive operators bind left to right.

3.5.1. *expression* + *expression*

The result is the sum of the two expressions. The scale of the result is the maximun of the scales of the expressions.

3.5.2. *expression* – *expression*

The result is the difference of the two expressions. The scale of the result is the maximum of the scales of the expressions.

3.6. assignment operators

The assignment operators bind right to left.

3.6.1. *named-expression* = *expression*

This expression results in assigning the value of the expression on the right to the named expression on the left.

3.6.2. *named-expression* =+ *expression*

3.6.3. *named-expression* =– *expression*

3.6.4. *named-expression* =* *expression*

3.6.5. *named-expression* =/ *expression*

3.6.6. *named-expression* =% *expression*

3.6.7. *named-expression* =^ *expression*

The result of the above expressions is equivalent to "named expression = named expression OP expression", where OP is the operator after the = sign.

4. Relations

Unlike all other operators, the relational operators are only valid as the object of an **if**, **while**, or inside a **for** statement.

4.1. *expression* < *expression*

4.2. *expression* > *expression*

4.3. *expression* <= *expression*

4.4. *expression* >= *expression*

4.5. *expression* == *expression*

4.6. *expression* != *expression*

5. Storage classes

There are only two storage classes in BC, global and automatic (local). Only identifiers that are to be local to a function need be declared with the **auto** command. The arguments to a function are local to the function. All other identifiers are assumed to be global and available to all

functions. All identifiers, global and local, have initial values of zero. Identifiers declared as **auto** are allocated on entry to the function and released on returning from the function. They therefore do not retain values between function calls. **auto** arrays are specified by the array name followed by empty square brackets.

Automatic variables in BC do not work in exactly the same way as in either C or PL/I. On entry to a function, the old values of the names that appear as parameters and as automatic variables are pushed onto a stack. Until return is made from the function, reference to these names refers only to the new values.

6. Statements

Statements must be separated by semicolon or newline. Except where altered by control statements, execution is sequential.

6.1. Expression statements

When a statement is an expression, unless the main operator is an assignment, the value of the expression is printed, followed by a newline character.

6.2. Compound statements

Statements may be grouped together and used when one statement is expected by surrounding them with { }.

6.3. Quoted string statements

"any string"

This statement prints the string inside the quotes.

6.4. If statements

if (*relation*) *statement*

The substatement is executed if the relation is true.

6.5. While statements

while (*relation*) *statement*

The statement is executed while the relation is true. The test occurs before each execution of the statement.

6.6. For statements

for (*expression*; *relation*; *expression*) *statement*

The for statement is the same as

first-expression
while (*relation*) {
    statement
    last-expression
}

All three expressions must be present.

6.7. Break statements

break

break causes termination of a **for** or **while** statement.

6.8. Auto statements

auto *identifier*[,*identifier*]

The auto statement causes the values of the identifiers to be pushed down. The identifiers can be ordinary identifiers or array identifiers. Array identifiers are specified by following the array name by empty square brackets. The auto statement must be the first statement in a function definition.

6.9. Define statements

define([*parameter*[, *parameter*...]]) {
statements}

The define statement defines a function. The parameters may be ordinary identifiers or array names. Array names must be followed by empty square brackets.

6.10. Return statements

return

return(*expression* **)**

The return statement causes termination of a function, popping of its auto variables, and specifies the result of the function. The first form is equivalent to **return(0)**. The result of the function is the result of the expression in parentheses.

6.11. Quit

The quit statement stops execution of a BC program and returns control to UNIX when it is first encountered. Because it is not treated as an executable statement, it cannot be used in a function definition or in an **if, for,** or **while** statement.

PROC286™ Software Support and the Dosc Command under ICON/UXB™

Mark Muhlestein

ICON INTERNATIONAL, INC.
October 10, 1986

This document describes the implementation of *dosc* and related software for the ICON computer systems. It is intended to assist developers and system administrators to understand the interface between ICON/UXB and the ICON/DOS environment.[1]

1. Hardware

The PROC286 has a 16K message buffer which is addressable from both the PROC286 and the DCP (Disk Cache Processor). This message buffer is used for parameter and data buffers which support terminal and disk I/O from the DCP. The message buffer is fully dual-ported and supports indivisible read-modify-write accesses. The message buffer starts at 0xDC000000 in the PROC286 address space. The PROC286 and the DCP are also capable of interrupting each other.

2. Software

2.1. Character device support

The first half of the message buffer is dedicated to character silos which handle the transfer of character data to and from the PROC286. This memory is divided into eight slots which provide bi-directional character I/O capability for up to eight MultiLink™ partitions. There are two silos for each of the eight slots, one for input to the PROC286 one for output. Each silo has a header which contains the insert and extract pointers within the silo. A version of MultiLink is provided which reads and writes these silos. If the consumer side of a silo is unable to remove the data fast enough, the producer side must wait for space to become available before inserting more data into the silo.

ICON/UXB supports access to the silos as terminal (tty) devices. They are typically called /dev/mtty0 through /dev/mtty7. They behave as ordinary terminal devices for all ICON/UXB software. Data written to mtty lines appears in the PROC286 input silos, and data placed in PROC286 output silos may be read by reading the appropriate mtty line.

The *dosc* program simply opens one of the mtty lines and then listens to the user's terminal for keyboard input, as well as listening to the mtty line for MultiLink output. All keyboard input is scanned to watch for exit and suspend sequences, then written to the mtty line. All mtty input (MultiLink output) is simply displayed on the user's terminal.

---

[1] The ICON/DOS package includes a licensed version of MS-DOS™ from Microsoft, and other ICON-developed support programs. The term "DOS" will be used throughout to refer generically to the ICON/DOS environment and services provided by MS-DOS.

Unless the user specifies a partition, the first thing *dosc* has to do is find an available partition. The directory /usr/spool/uucp is scanned to find the first available partition which does not have a lock file. The lock files are named LCK..mtty*n*, and *dosc* creates one exclusively for the mtty line it opens. If, for some reason, a *dosc* process is killed and does not exit normally, the lock file may be left in /usr/spool/uucp. Until the lock file is removed, *dosc* will not access the corresponding mtty line. During system restart all lock files are automatically removed by /etc/rc.

The file /etc/mttys should contain with the number of MultiLink partitions which have been started. If all MultiLink partitions are busy, the user is directed to try again later.

When *dosc* initializes, it first determines the terminal type by examining the environment variable TERM.[2] The terminal is sent an initialization sequence which causes the screen to blank then repaint the MultiLink partition's screen image. Once *dosc* has initialized the terminal, the terminal keyboard sends make-break scan codes. This means that if *dosc* is not able to send the reset command to the terminal before exiting (the system went down or the *dosc* process was killed), the terminal will send unintelligible scan codes to the ICON/UXB environment. To correct this situation, it is necessary to power the terminal off then back on, then, if the user is still logged in, type 'reset^J' (control-J) to reset the *stty* parameters to something sensible. If the reset command doesn't seem to work, try it again. As a last resort, kill the user's shell process.

There are two ways to return from the DOS environment to the ICON/UXB environment: exit and suspend. See the manual entry for *dosc(1)* for the exact key sequence to use for each. The exit sequence causes *dosc* to reset the keyboard to normal ascii mode, remove the lock file, and exit. Because the lock file is removed, *dosc* will be successful the next time it attempts to access the corresponding partition. This means that the user will have the same DOS environment the original user had when he left. For this reason, it may not be advisable to use the exit command sequence while in an DOS application because it may be confusing to the next user of that partition.

The suspend sequence causes *dosc* to reset the keyboard to normal ascii mode, but it leaves the lock file in place. It then sends itself a STOP signal which suspends execution and returns to *csh*. The user may then use any desired ICON/UXB commands without danger of his DOS session being interfered with. When the user is ready to continue the DOS session, he may do so by using the 'fg' command of *csh*. If the user has other background jobs he can do a 'jobs' command to see which one to 'fg' back to.

2.2. Disk support

The other half of the message buffer is dedicated to handling disk requests from DOS. There is a small parameter block at the start of the disk section, followed by seventeen 512 byte buffers. DOS never requests more than seventeen contiguous blocks, so all requests can be satisfied n one transfer. The PROC286 BIOS has been implemented to support disk type 14 as an "ICON/UXB" type disk. All DOS requests to type 14 disks are routed through the PROC286 message buffer and access specially designated partitions on hard disks which are physically connected to the DCP.

---

[2] The current version supports fully only one type of terminal: the ICON DT1200™ Data Terminal. Other types of terminals may be supported, but they may have problems interacting with MultiLink. See the MultiLink Advanced manual for a discussion of support for other terminals.

DT1200 is a trademark of ICON INTERNATIONAL, INC.

Part of the function of the DCP is to periodically examine the PROC286 message buffer for disk requests. Since the BIOS interface is at the block level, I/O requests are in the form of a read or write of a specified number of blocks, starting with a specified block number.

Currently the DOS disks are assigned as follows: If the PROC286 setup indicates that drive 0 is other than type 14, the BIOS assumes a real PC-type disk is present and it assigns it as drive C. If drive 0 is type 14, it is assigned to the first ICON/UXB virtual disk.[3] If drive 1 is type 14, it is assigned to the next available ICON/UXB virtual disk. It is currently not possible to have the ICON/UXB disk as drive C and a real PC-type disk as drive D. **It is imperative that any PROC286 type 14 drives have an ICON/UXB virutal disk defined for them.**

Once the ICON/UXB system is up, DOS initialization of type 14 disks and installation of DOS software may proceed normally (fdisk, format, etc.). Disk access is transparent to programs which use the BIOS for disk requests except that file accesses are likely to be much faster because the DCP uses its cache memory for servicing the requests.

One of the difficult issues with using the above-described interface is that certain popular DOS-compatible programs make direct access to the disk hardware for the purpose of enforcing copy protection schemes. Obviously, if no disk hardware is present on the PROC286 these schemes are doomed to failure. Users should be aware that it is possible to destroy distribution diskettes by attempting to install the copy protected software onto a type 14 disk. The application's install program copies the software to the hard disk, issues commands to the (non-existent) hard disk controller, then disables the installation diskette from further use. Any method which allows the application to be copied will permit it to be successfully installed.

2.3. Special utilities to facilitate ICON/UXB—DOS synergy (or at least, peaceful coexistence).

This section describes the programs available under DOS which can be used to facilitate access to ICON/UXB resources, as well as to allow ICON/UXB to have access to DOS files. These utilities use the UNIX*n* and SLPT*n* DOS device drivers to implement a data path between the two systems.

2.3.1. The UNIX*n* DOS device driver

The config.sys file supplied by ICON should have an entry for **device=UNIX.DEV** in it. This interface supports eight character devices, UNIX0 through UNIX7. These devices correspond to /dev/mtty0 through /dev/mtty7 on the ICON/UXB side, and may be used to write software to communicate between the processors. Because the data path is totally internal to the machine, the data transfer rates are adequate for most file transfer purposes. Greater speed can be achieved by directly manipulating the PROC286 message buffer, at the cost of somewhat increased application program complexity. Contact ICON for further information.

---

[3] See *dosdisk(8)* and *dosdisks(5)*.

2.3.2. The SLPT*n* DOS device driver

This driver permits the redirection of DOS printer output to the ICON/UXB spooling system. Eight virtual spooled devices are supported, SLPT0 through SLPT7. Data written to these devices is read by a server process (*dosprint*) on the ICON/UXB system through one of the /dev/mtty devices (/dev/mtty6 by default).

The config.sys entry for this driver has the following form:

device=SLPT.DEV [mtty*n*] [LPT1>SLPT*n1*] [LPT2>SLPT*n2*] [LPT3>SLPT*n3*]

where items enclosed in square brackets are optional.

The first parameter allows overriding the mtty line used for communication with ICON/UXB. The default value of mtty6 should be used unless it is necessary to use the corresponding MultiLink partition for some reason. If it becomes necessary to change this parameter, the correct argument to the ICON/UXB *dosprint* program (described below) must be supplied.

The other three optional parameters may be used to intercept output which would normally go to the standard DOS parallel printers and redirect it to the desired SLPT printer for eventual printing by the ICON/UXB spooler. For example, the line

device=SLPT.DEV LPT1>SLPT0 LPT2>SLPT5

in the config.sys file will cause all output for the LPT1 printer to go to SLPT0, the output for the LPT2 printer to go to SLPT5, and the output for the LPT3 printer (if any) to go to the actual DOS printer.

All output to the SLPT devices is spooled into files on the ICON/UXB file system. The spool files are created in the /usr/spool/dos directory. If two MultiLink partitions direct output to the same printer two different spool files will be created. It is not necessary to use the MultiLink spooler when using the SLPT driver.[4]

The *dosprint* program runs as an ICON/UXB "daemon" server process which is initiated as part of the startup procedure. Its function is to interpret the data stream from the SLPT driver. This data has header information which tells which MultiLink partition and SLPT printer generated that data. As data is received it is sorted and output to the correct spool file. Every 10 seconds the *dosprint* program checks to see which active spool files have received data. If two consecutive checks pass without data being sent to a given spool file, that file is closed and all accumulated data is queued for printing using the ICON/UXB *lpr* command. The interval between checks for active spool files may be adjusted from 10 seconds if necessary (e.g. on a heavily loaded system).

The file /etc/dosprinters is provided in order to route the output from a specific SLPT print device to the desired ICON/UXB spooled printer. This file consists of zero or more lines in the following format:

x pr [opt]

where **x** is 0-7 corresponding to SLPT0-7, **pr** is the ICON/UXB printer to receive the output, and **[opt]** is an optional string which is passed to *lpr* which can be used to set various modes. For example,

---

[4] Actually, there appears to be an incompatibility in the current release of MultiLink with the DOS *PRINT* command. Since background printing is available through either the SLPT driver or MultiLink spooling, this is not seen as a serious problem.

```
1 lp
3 lp -p -h
7 icanthor
```

specifies that the output from SLPT1 should be spooled to "lp" (this is actually the default); the output from SLPT3 is spooled to lp with the -p and -h flags (which causes *lpr* to pass the file through the *pr* filter and print it without the header); and the output from SLPT7 is spooled to a printer known in /etc/printcap as "icanthor". If the printer specifies a remote destination (that is, on another node on the ICON/UXB network), the ICON/UXB spooler will automatically forward print files to that node. Notice that it is not necessary to specify an entry for all eight printers; all SLPT devices default to "lp" with no options. See the manual entries for *printcap(5)* and *lpr(1)* in the *MPS/UX Reference Manual* for further information on setting up the ICON/UXB spooler.

2.3.3. The UCOPY program

UCOPY is an DOS program that allows users to transfer data between the DOS and ICON/UXB environments. Because peripheral devices are accessed as files in both environments it is also possible to copy data to and from those peripheral devices. *UCOPY* communicates to an ICON/UXB server process (*doscopyd*) through the /dev/mtty7 port. This device is currently reserved for use by DOS utility programs which use *doscopyd*.

The *UCOPY* command syntax is

UCOPY[/A] :[user]:pathname1 pathname2
or
UCOPY[/A] pathname1 :[user]:pathname2

In both cases, **pathname1** is copied to **pathname2** with the ':' indicating the ICON/UXB pathname. Between the colons the user may supply an optional ICON/UXB user name to be used for the copy. If no user name is given, the default is the user name of the *dosc* process. If the command is given from the DOS main console (which is not logged in under ICON/UXB) a user name must be supplied.

The **pathname1** and **pathname2** arguments must be specified as files on both systems; they may not be directory names. Either forward or backward slashes may be used to specify pathnames for either argument; *UCOPY* converts them to the appropriate type for each environment.

The DOS environment variable **UDIR** may be set to a directory pathname in the ICON/UXB file structure. **UDIR** will be prepended to the ICON/UXB pathname argument if the argument does not begin with a slash. This allows several files to be copied to or from the same ICON/UXB directory without having to specify the directory every time *UCOPY* is invoked.

If the ICON/UXB *doscopyd* daemon verifies that the permissions are correct, the source file exists, etc., the transfer is made. If the ICON/UXB file can not be accessed or created because the ICON/UXB user name is not authorized to perform the action, or because a specified user name requires a password, *UCOPY* prompts the user for a new user name and password. This is continued until the user supplies a valid user name and password, or *UCOPY* runs out of patience.

The **/A** switch should be used for copying text type files. It specifies that when transferring from ICON/UXB to DOS, ascii CR characters are to be inserted at the end

of each line. When copying from DOS to ICON/UXB all ascii CR characters are removed from the data stream, and copying stops if a ^Z is encountered. The **/A** option thus converts text files to the appropriate standard representation for each of the systems.

Application-specific conversion packages are available which allow various database, word processing, etc., file formats to be utilized on both systems.

Currently the *UCOPY* functionality is only available from the DOS environment, and it allows only one user to do file transfers at a time. This is because of the single-threaded nature of DOS. In the future, ICON may provide a server running in a MultiLink partition to allow handling multiple concurrent copies, as well as copy requests from ICON/UXB users.

2.3.4. The TAR utility for DOS

The DOS version of *TAR* provides the user in the DOS environment the ability to back up and restore files and directory structures. It uses the same format as the ICON/UXB tar program for data storage and *tar* files may be freely interchanged between the two environments. *TAR* uses the ICON/UXB *doscopyd* server process to copy data to and from the backup medium. Any suitable ICON/UXB file or device may be used. The command syntax for DOS *TAR* is the same as that used in ICON/UXB except that the **r** and **u** options are not supported.[5] *TAR* may be used to copy or move a directory structure to another part of the DOS file system without having to copy each directory and subdirectory individually. Individual files may also be extracted from the backup.

2.3.5. Whodos

Whodos provides a user in the ICON/UXB environment the ability to monitor *dosc* users and determine which MultiLink partitions are in use and by whom. It can be useful when all partitions are in use, or a specific partition is needed and it is in use. It is invoked from the ICON/UXB environment simply by entering the command

whodos

from the shell. The number of partitions available is determined from the file /etc/mttys.

---

[5] The subset of TAR options supported is not yet finalized.

JOVE Manual for UNIX Users

Jonathan Payne
(revised for 4.3BSD by Doug Kingston and Mark Seiden)

1. Introduction

JOVE* is an advanced, self-documenting, customizable real-time display editor. It (and this tutorial introduction) are based on the original EMACS editor and user manual written at M.I.T. by Richard Stallman+.

JOVE is considered a *display* editor because normally the text being edited is visible on the screen and is updated automatically as you type your commands.

It's considered a *real-time* editor because the display is updated very frequently, usually after each character or pair of characters you type. This minimizes the amount of information you must keep in your head as you edit.

JOVE is *advanced* because it provides facilities that go beyond simple insertion and deletion: filling of text; automatic indentations of programs; view more than one file at once; and dealing in terms of characters, words, lines, sentences and paragraphs. It is much easier to type one command meaning "go to the end of the paragraph" than to find the desired spot with repetition of simpler commands.

Self-documenting means that at almost any time you can easily find out what a command does, or to find all the commands that pertain to a topic.

Customizable means that you can change the definition of JOVE commands in little ways. For example, you can rearrange the command set; if you prefer to use arrow keys for the four basic cursor motion commands (up, down, left and right), you can. Another sort of customization is writing new commands by combining built in commands.

2. The Organization of the Screen

JOVE divides the screen up into several sections. The biggest of these sections is used to display the text you are editing. The terminal's cursor shows the position of *point*, the location at which editing takes place. While the cursor appears to point *at* a character, point should be thought of as between characters; it points *before* the character that the cursor appears on top of. Terminals have only one cursor, and when output is in progress it must appear where the typing is being done. This doesn't mean that point is moving; it is only that JOVE has no way of showing you the location of point except when the terminal is idle.

The lines of the screen are usually available for displaying text but sometimes are pre-empted by typeout from certain commands (such as a listing of all the editor commands). Most of the time, output from commands like these is only desired for a short period of time, usually just long enough to glance at it. When you have finished looking at the output, you can type Space to make your text reappear. (Usually a Space that you type inserts itself, but when there is typeout on the screen, it does nothing but get rid of that). Any other command executes normally, *after* redrawing your text.

2.1. The Message Line

The bottom line on the screen, called the *message line*, is reserved for printing messages and for accepting input from the user, such as filenames or search strings. When JOVE prompts for input, the cursor will temporarily appear on the bottom line, waiting for you to type a string. When you have finished typing your input, you can type a Return to send it to JOVE. If you change your mind about running the command that is waiting for input, you can type Control-G to abort, and you can continue with your editing.

---

*JOVE stands for Jonathan's Own Version of Emacs.

+Although JOVE is meant to be compatible with EMACS, and indeed many of the basic commands are very similar, there are some major differences between the two editors, and you should not rely on their behaving identically.

When JOVE is prompting for a filename, all the usual editing facilities can be used to fix typos and such; in addition, JOVE has the following extra functions:

^N Insert the next filename from the argument list.

^P Insert the previous filename from the argument list.

^R Insert the full pathname of the file in the current buffer.

Sometimes you will see **--more--** on the message line. This happens when typeout from a command is too long to fit in the screen. It means that if you type a Space the next screenful of typeout will be printed. If you are not interested, typing anything but a Space will cause the rest of the output to be discarded. Typing C-G will discard the output and print *Aborted* where the **--more--** was. Typing any other command will discard the rest of the output and also execute the command.

The message line and the list of filenames from the shell command that invoked JOVE are kept in a special buffer called *Minibuf* that can be edited like any other buffer.

2.2. The Mode Line

At the bottom of the screen, but above the message line, is the *mode line*. The mode line format looks like this:

JOVE (major minor) Buffer: bufr "file" *

major is the name of the current *major mode*. At any time, JOVE can be in only one major mode at a time. Currently there are only four major modes: *Fundamental*, *Text*, *Lisp* and *C*.

minor is a list of the minor modes that are turned on. **Abbrev** means that *Word Abbrev* mode is on; **AI** means that *Auto Indent* mode is on; **Fill** means that *Auto Fill* mode is on; **OvrWt** means that *Over Write* mode is on. **Def** means that you are in the process of defining a keyboard macro. This is not really a mode, but it's useful to be reminded about it. The meanings of these modes are described later in this document.

bufr is the name of the currently selected *buffer*. Each buffer has its own name and holds a file being edited; this is how JOVE can hold several files at once. But at any given time you are editing only one of them, the *selected* buffer. When we speak of what some command does to "the buffer", we are talking about the currently selected buffer. Multiple buffers makes it easy to switch around between several files, and then it is very useful that the mode line tells you which one you are editing at any time. (You will see later that it is possible to divide the screen into multiple *windows*, each showing a different buffer. If you do this, there is a mode line beneath each window.)

file is the name of the file that you are editing. This is the default filename for commands that expect a filename as input.

The asterisk at the end of the mode line means that there are changes in the buffer that have not been saved in the file. If the file has not been changed since it was read in or saved, there is no asterisk.

3. Command Input Conventions

3.1. Notational Conventions for ASCII Characters

In this manual, "Control" characters (that is, characters that are typed with the Control key and some other key at the same time) are represented by "C-" followed by another character. Thus, C-A is the character you get when you type A with the Control key (sometimes labeled CTRL) down. Most control characters when present in the JOVE buffer are displayed with a caret; thus, ^A for C-A. Rubout (or DEL) is displayed as ^?, escape as ^[.

3.2. Command and Filename Completion

When you are typing the name of a JOVE command, you need type only enough letters to make the name unambiguous. At any point in the course of typing the name, you can type question mark (?) to see a list of all the commands whose names begin with the characters you've already typed; you can type Space to have JOVE supply as many characters as it can; or you can type Return to complete the command if there

is only one possibility. For example, if you have typed the letters "*au*" and you then type a question mark, you will see the list

auto-execute-command
auto-execute-macro
auto-fill-mode
auto-indent-mode

If you type a Return at this point, JOVE will complain by ringing the bell, because the letters you've typed do not unambiguously specify a single command. But if you type Space, JOVE will supply the characters "*to-*" because all commands that begin "*au*" also begin "*auto-*". You could then type the letter "*f*" followed by either Space or Return, and JOVE would complete the entire command.

Whenever JOVE is prompting you for a filename, say in the *find-file* command, you also need only type enough of the name to make it unambiguous with respect to files that already exist. In this case, question mark and Space work just as they do in command completion, but Return always accepts the name just as you've typed it, because you might want to create a new file with a name similar to that of an existing file.

4. Commands and Variables

JOVE is composed of *commands* which have long names such as *next-line*. Then *keys* such as C-N are connected to commands through the *command dispatch table*. When we say that C-N moves the cursor down a line, we are glossing over a distinction which is unimportant for ordinary use, but essential for simple customization: it is the command *next-line* which knows how to move a down line, and C-N moves down a line because it is connected to that command. The name for this connection is a *binding*; we say that the key C-N *is bound to* the command *next-line*.

Not all commands are bound to keys. To invoke a command that isn't bound to a key, you can type the sequence ESC X, which is bound to the command *execute-named-command*. You will then be able to type the name of whatever command you want to execute on the message line.

Sometimes the description of a command will say "to change this, set the variable *mumble-foo*". A variable is a name used to remember a value. JOVE contains variables which are there so that you can change them if you want to customize. The variable's value is examined by some command, and changing that value makes the command behave differently. Until you are interesting in customizing JOVE, you can ignore this information.

4.1. Prefix Characters

Because there are more command names than keys, JOVE provides *prefix characters* to increase the number of commands that can be invoked quickly and easily. When you type a prefix character JOVE will wait for another character before deciding what to do. If you wait more than a second or so, JOVE will print the prefix character on the message line as a reminder and leave the cursor down there until you type your next character. There are two prefix characters built into JOVE: Escape and Control-X. How the next character is interpreted depends on which prefix character you typed. For example, if you type Escape followed by B you'll run *backward-word*, but if you type Control-X followed by B you'll run *select-buffer*. Elsewhere in this manual, the Escape key is indicated as "ESC", which is also what JOVE displays on the message line for Escape.

4.2. Help

To get a list of keys and their associated commands, you type ESC X *describe-bindings*. If you want to describe a single key, ESC X *describe-key* will work. A description of an individual command is available by using ESC X *describe-command*, and descriptions of variables by using ESC X *describe-variable*. If you can't remember the name of the thing you want to know about, ESC X *apropos* will tell you if a command or variable has a given string in its name. For example, ESC X *apropos describe* will list the names of the four describe commands mentioned briefly in this section.

5. Basic Editing Commands

5.1. Inserting Text

To insert printing characters into the text you are editing, just type them. All printing characters you type are inserted into the text at the cursor (that is, at *point*), and the cursor moves forward. Any characters after the cursor move forward too. If the text in the buffer is FOOBAR, with the cursor before the B, then if you type XX, you get FOOXXBAR, with the cursor still before the B.

To correct text you have just inserted, you can use Rubout. Rubout deletes the character *before* the cursor (not the one that the cursor is on top of or under; that is the character *after* the cursor). The cursor and all characters after it move backwards. Therefore, if you typing a printing character and then type Rubout, they cancel out.

To end a line and start typing a new one, type Return. Return operates by inserting a *line-separator*, so if you type Return in the middle of a line, you break the line in two. Because a line-separator is just a single character, you can type Rubout at the beginning of a line to delete the line-separator and join it with the preceding line.

As a special case, if you type Return at the end of a line and there are two or more empty lines just below it, JOVE does not insert a line-separator but instead merely moves to the next (empty) line. This behavior is convenient when you want to add several lines of text in the middle of a buffer. You can use the Control-O (*newline-and-backup*) command to "open" several empty lines at once; then you can insert the new text, filling up these empty lines. The advantage is that JOVE does not have to redraw the bottom part of the screen for each Return you type, as it would ordinarily. That "redisplay" can be both slow and distracting.

If you add too many characters to one line, without breaking it with Return, the line will grow too long to display on one screen line. When this happens, JOVE puts an "!" at the extreme right margin, and doesn't bother to display the rest of the line unless the cursor happens to be in it. The "!" is not part of your text; conversely, even though you can't see the rest of your line, it's still there, and if you break the line, the "!" will go away.

Direct insertion works for printing characters and space, but other characters act as editing commands and do not insert themselves. If you need to insert a control character, Escape, or Rubout, you must first *quote* it by typing the Control-Q command first.

5.2. Moving the Cursor

To do more than insert characters, you have to know how to move the cursor. Here are a few of the commands for doing that.

C-A	Move to the beginning of the line.
C-E	Move to the end of the line.
C-F	Move forward over one character.
C-B	Move backward over one character.
C-N	Move down one line, vertically. If you start in the middle of one line, you end in the middle of the next.
C-P	Move up one line, vertically.
ESC <	Move to the beginning of the entire buffer.
ESC >	Move to the end of the entire buffer.
ESC ,	Move to the beginning of the visible window.
ESC .	Move to the end of the visible window.

5.3. Erasing Text

Rubout	Delete the character before the cursor.
C-D	Delete the character after the cursor.
C-K	Kill to the end of the line.

You already know about the Rubout command which deletes the character before the cursor. Another command, Control-D, deletes the character after the cursor, causing the rest of the text on the line to shift left. If Control-D is typed at the end of a line, that line and the next line are joined together.

To erase a larger amount of text, use the Control-K command, which kills a line at a time. If Control-K is done at the beginning or middle of a line, it kills all the text up to the end of the line. If Control-K is done at the end of a line, it joins that line and the next line. If Control-K is done twice, it kills the rest of the line and the line separator also.

5.4. Files — Saving Your Work

The commands above are sufficient for creating text in the JOVE buffer. The more advanced JOVE commands just make things easier. But to keep any text permanently you must put it in a *file*. Files are the objects which UNIX† uses for storing data for a length of time. To tell JOVE to read text into a file, choose a filename, such as *foo.bar*, and type C-X C-R *foo.bar*<return>. This reads the file *foo.bar* so that its contents appear on the screen for editing. You can make changes, and then save the file by typing C-X C-S (save-file). This makes the changes permanent and actually changes the file *foo.bar*. Until then, the changes are only inside JOVE, and the file *foo.bar* is not really changed. If the file *foo.bar* doesn't exist, and you want to create it, read it as if it did exist. When you save your text with C-X C-S the file will be created.

5.5. Exiting and Pausing — Leaving JOVE

The command C-X C-C (*exit-jove*) will terminate the JOVE session and return to the shell. If there are modified but unsaved buffers, JOVE will ask you for confirmation, and you can abort the command, look at what buffers are modified but unsaved using C-X C-B (*list-buffers*), save the valuable ones, and then exit. If what you want to do, on the other hand, is *preserve* the editing session but return to the shell temporarily you can (under Berkeley UNIX only) issue the command ESC S (*pause-jove*), do your UNIX work within the c-shell, then return to JOVE using the *fg* command to resume editing at the point where you paused. For this sort of situation you might consider using an *interactive shell* (that is, a shell in a JOVE window) which lets you use editor commands to manipulate your UNIX commands (and their output) while never leaving the editor. (The interactive shell feature is described below.)

5.6. Giving Numeric Arguments to JOVE Commands

Any JOVE command can be given a *numeric argument*. Some commands interpret the argument as a repetition count. For example, giving an argument of ten to the C-F command (forward-character) moves forward ten characters. With these commands, no argument is equivalent to an argument of 1.

Some commands use the value of the argument, but do something peculiar (or nothing) when there is no argument. For example, ESC G (*goto-line*) with an argument **n** goes to the beginning of the **n**'th line. But ESC G with no argument doesn't do anything. Similarly, C-K with an argument kills that many lines, including their line separators. Without an argument, C-K when there is text on the line to the right of the cursor kills that text; when there is no text after the cursor, C-K deletes the line separator.

The fundamental way of specifying an argument is to use ESC followed by the digits of the argument, for example, ESC 123 ESC G to go to line 123. Negative arguments are allowed, although not all of the commands know what to do with one.

Typing C-U means do the next command four times. Two such C-U's multiply the next command by sixteen. Thus, C-U C-U C-F moves forward sixteen characters. This is a good way to move forward quickly,

† UNIX is a trademark of Bell Laboratories.

since it moves about 1/4 of a line on most terminals. Other useful combinations are: C-U C-U C-N (move down a good fraction of the screen), C-U C-U C-O (make "a lot" of blank lines), and C-U C-K (kill four lines — note that typing C-K four times would kill 2 lines).

There are other, terminal-dependent ways of specifying arguments. They have the same effect but may be easier to type. If your terminal has a numeric keypad which sends something recognizably different from the ordinary digits, it is possible to program JOVE to to allow use of the numeric keypad for specifying arguments.

5.7. The Mark and the Region

In general, a command that processes an arbitrary part of the buffer must know where to start and where to stop. In JOVE, such commands usually operate on the text between point and *the mark*. This body of text is called *the region*. To specify a region, you set point to one end of it and mark at the other. It doesn't matter which one comes earlier in the text.

C-@ Set the mark where point is.

C-X C-X Interchange mark and point.

For example, if you wish to convert part of the buffer to all upper-case, you can use the C-X C-U command, which operates on the text in the region. You can first go to the beginning of the text to be capitalized, put the mark there, move to the end, and then type C-X C-U. Or, you can set the mark at the end of the text, move to the beginning, and then type C-X C-U. C-X C-U runs the command *case-region-upper*, whose name signifies that the region, or everything between point and mark, is to be capitalized.

The way to set the mark is with the C-@ command or (on some terminals) the C-Space command. They set the mark where point is. Then you can move point away, leaving mark behind. When the mark is set, "[Point pushed]" is printed on the message line.

Since terminals have only one cursor, there is no way for JOVE to show you where the mark is located. You have to remember. The usual solution to this problem is to set the mark and then use it soon, before you forget where it is. But you can see where the mark is with the command C-X C-X which puts the mark where point was and point where mark was. The extent of the region is unchanged, but the cursor and point are now at the previous location of the mark.

5.8. The Ring of Marks

Aside from delimiting the region, the mark is also useful for remembering a spot that you may want to go back to. To make this feature more useful, JOVE remembers 16 previous locations of the mark. Most commands that set the mark push the old mark onto this stack. To return to a marked location, use C-U C-@. This moves point to where the mark was, and restores the mark from the stack of former marks. So repeated use of this command moves point to all of the old marks on the stack, one by one. Since the stack is actually a ring, enough uses of C-U C-@ bring point back to where it was originally.

Some commands whose primary purpose is to move point a great distance take advantage of the stack of marks to give you a way to undo the command. The best example is ESC <, which moves to the beginning of the buffer. If there are more than 22 lines between the beginning of the buffer and point, ESC < sets the mark first, so that you can use C-U C-@ or C-X C-X to go back to where you were. You can change the number of lines from 22 since it is kept in the variable *mark-threshold*. By setting it to 0, you can make these commands always set the mark. By setting it to a very large number you can prevent these commands from ever setting the mark. If a command decides to set the mark, it prints the message */Point pushed/*.

5.9. Killing and Moving Text

The most common way of moving or copying text with JOVE is to kill it, and get it back again in one or more places. This is very safe because the last several pieces of killed text are all remembered, and it is versatile, because the many commands for killing syntactic units can also be used for moving those units. There are also other ways of moving text for special purposes.

5.10. Deletion and Killing

Most commands which erase text from the buffer save it so that you can get it back if you change your mind, or move or copy it to other parts of the buffer. These commands are known as *kill* commands. The rest of the commands that erase text do not save it; they are known as *delete* commands. The delete commands include C-D and Rubout, which delete only one character at a time, and those commands that delete only spaces or line separators. Commands that can destroy significant amounts of nontrivial data generally kill. A command's name and description will use the words *kill* or *delete* to say which one it does.

C-D	Delete next character.
Rubout	Delete previous character.
ESC \	Delete spaces and tabs around point.
C-X C-O	Delete blank lines around the current line.
C-K	Kill rest of line or one or more lines.
C-W	Kill region (from point to the mark).
ESC D	Kill word.
ESC Rubout	Kill word backwards.
ESC K	Kill to end of sentence.
C-X Rubout	Kill to beginning of sentence.

5.11. Deletion

The most basic delete commands are C-D and Rubout. C-D deletes the character after the cursor, the one the cursor is "on top of" or "underneath". The cursor doesn't move. Rubout deletes the character before the cursor, and moves the cursor back. Line separators act like normal characters when deleted. Actually, C-D and Rubout aren't always *delete* commands; if you give an argument, they *kill* instead. This prevents you from losing a great deal of text by typing a large argument to a C-D or Rubout.

The other delete commands are those which delete only formatting characters: spaces, tabs, and line separators. ESC \ (*delete-white-space*) deletes all the spaces and tab characters before and after point. C-X C-O (*delete-blank-lines*) deletes all blank lines after the current line, and if the current line is blank deletes all the blank lines preceding the current line as well (leaving one blank line, the current line).

5.12. Killing by Lines

The simplest kill command is the C-K command. If issued at the beginning of a line, it kills all the text on the line, leaving it blank. If given on a line containing only white space (blanks and tabs) the line disappears. As a consequence, if you go to the front of a non-blank line and type two C-K's, the line disappears completely.

More generally, C-K kills from point up to the end of the line, unless it is at the end of a line. In that case, it kills the line separator following the line, thus merging the next line into the current one. Invisible spaces and tabs at the end of the line are ignored when deciding which case applies, so if point appears to be at the end of the line, you can be sure the line separator will be killed.

C-K with an argument of zero kills all the text before point on the current line.

5.13. Other Kill Commands

A kill command which is very general is C-W (*kill-region*), which kills everything between point and the mark.* With this command, you can kill and save contiguous characters, if you first set the mark at one end of them and go to the other end.

Other syntactic units can be killed, too; words, with ESC Rubout and ESC D; and, sentences, with ESC K and C-X Rubout.

*Often users switch this binding from C-W to C-X C-K because it is too easy to hit C-W accidentally.

5.14. Un-killing

Un-killing (yanking) is getting back text which was killed. The usual way to move or copy text is to kill it and then un-kill it one or more times.

C-Y Yank (re-insert) last killed text.

ESC Y Replace re-inserted killed text with the previously killed text.

ESC W Save region as last killed text without killing.

Killed text is pushed onto a ***ring buffer*** called the ***kill ring*** that remembers the last 10 blocks of text that were killed. (Why it is called a ring buffer will be explained below). The command C-Y (*yank*) reinserts the text of the most recent kill. It leaves the cursor at the end of the text, and puts the mark at the beginning. Thus, a single C-Y undoes the C-W.

If you wish to copy a block of text, you might want to use ESC W (*copy-region*), which copies the region into the kill ring without removing it from the buffer. This is approximately equivalent to C-W followed by C-Y, except that ESC W does not mark the buffer as "changed" and does not cause the screen to be rewritten.

There is only one kill ring shared among all the buffers. After visiting a new file, whatever was last killed in the previous file is still on top of the kill ring. This is important for moving text between files.

5.15. Appending Kills

Normally, each kill command pushes a new block onto the kill ring. However, two or more kill commands immediately in a row (without any other intervening commands) combine their text into a single entry on the ring, so that a single C-Y command gets it all back as it was before it was killed. This means that you don't have to kill all the text in one command; you can keep killing line after line, or word after word, until you have killed it all, and you can still get it all back at once.

Commands that kill forward from *point* add onto the end of the previous killed text. Commands that kill backward from *point* add onto the beginning. This way, any sequence of mixed forward and backward kill commands puts all the killed text into one entry without needing rearrangement.

5.16. Un-killing Earlier Kills

To recover killed text that is no longer the most recent kill, you need the ESC Y (*yank-pop*) command. The ESC Y command can be used only after a C-Y (yank) command or another ESC Y. It takes the un-killed text inserted by the C-Y and replaces it with the text from an earlier kill. So, to recover the text of the next-to-the-last kill, you first use C-Y to recover the last kill, and then discard it by use of ESC Y to move back to the previous kill.

You can think of all the last few kills as living on a ring. After a C-Y command, the text at the front of the ring is also present in the buffer. ESC Y "rotates" the ring bringing the previous string of text to the front and this text replaces the other text in the buffer as well. Enough ESC Y commands can rotate any part of the ring to the front, so you can get at any killed text so long as it is recent enough to be still in the ring. Eventually the ring rotates all the way around and the most recently killed text comes to the front (and into the buffer) again. ESC Y with a negative argument rotates the ring backwards.

When the text you are looking for is brought into the buffer, you can stop doing ESC Y's and the text will stay there. It's really just a copy of what's at the front of the ring, so editing it does not change what's in the ring. And the ring, once rotated, stays rotated, so that doing another C-Y gets another copy of what you rotated to the front with ESC Y.

If you change your mind about un-killing, C-W gets rid of the un-killed text, even after any number of ESC Y's.

6. Searching

The search commands are useful for finding and moving to arbitrary positions in the buffer in one swift motion. For example, if you just ran the spell program on a paper and you want to correct some word, you can use the search commands to move directly to that word. There are two flavors of search: *string*

search and *incremental search*. The former is the default flavor—if you want to use incremental search you must rearrange the key bindings (see below).

6.1. Conventional Search

C-S Search forward.

C-R Search backward.

To search for the string "FOO" you type "C-S FOO<return>". If JOVE finds FOO it moves point to the end of it; otherwise JOVE prints an error message and leaves point unchanged. C-S searches forward from point so only occurrences of FOO after point are found. To search in the other direction use C-R. It is exactly the same as C-S except it searches in the opposite direction, and if it finds the string, it leaves point at the beginning of it, not at the end as in C-S.

While JOVE is searching it prints the search string on the message line. This is so you know what JOVE is doing. When the system is heavily loaded and editing in exceptionally large buffers, searches can take several (sometimes many) seconds.

JOVE remembers the last search string you used, so if you want to search for the same string you can type "C-S <return>". If you mistyped the last search string, you can type C-S followed by C-R. C-R, as usual, inserts the default search string into the minibuffer, and then you can fix it up.

6.2. Incremental Search

This search command is unusual in that is is *incremental*; it begins to search before you have typed the complete search string. As you type in the search string, JOVE shows you where it would be found. When you have typed enough characters to identify the place you want, you can stop. Depending on what you will do next, you may or may not need to terminate the search explicitly with a Return first.

The command to search is C-S (*i-search-forward*). C-S reads in characters and positions the cursor at the first occurrence of the characters that you have typed so far. If you type C-S and then F, the cursor moves in the text just after the next "F". Type an "O", and see the cursor move to after the next "FO". After another "O", the cursor is after the next "FOO". At the same time, the "FOO" has echoed on the message line.

If you type a mistaken character, you can rub it out. After the FOO, typing a Rubout makes the "O" disappear from the message line, leaving only "FO". The cursor moves back in the buffer to the "FO". Rubbing out the "O" and "F" moves the cursor back to where you started the search.

When you are satisfied with the place you have reached, you can type a Return, which stops searching, leaving the cursor where the search brought it. Also, any command not specially meaningful in searches stops the searching and is then executed. Thus, typing C-A would exit the search and then move to the beginning of the line. Return is necessary only if the next character you want to type is a printing character, Rubout, Return, or another search command, since those are the characters that have special meanings inside the search.

Sometimes you search for "FOO" and find it, but not the one you hoped to find. Perhaps there is a second FOO that you forgot about, after the one you just found. Then type another C-S and the cursor will find the next FOO. This can be done any number of times. If you overshoot, you can return to previous finds by rubbing out the C-S's.

After you exit a search, you can search for the same string again by typing just C-S C-S: one C-S command to start the search and then another C-S to mean "search again for the same string".

If your string is not found at all, the message line says "Failing I-search". The cursor is after the place where JOVE found as much of your string as it could. Thus, if you search for FOOT and there is no FOOT, you might see the cursor after the FOO in FOOL. At this point there are several things you can do. If your string was mistyped, you can rub some of it out and correct it. If you like the place you have found, you can type Return or some other JOVE command to "accept what the search offered". Or you can type C-G, which undoes the search altogether and positions you back where you started the search.

You can also type C-R at any time to start searching backwards. If a search fails because the place you started was too late in the file, you should do this. Repeated C-R's keep looking backward for more occurrences of the last search string. A C-S starts going forward again. C-R's can be rubbed out just like anything else.

6.3. Searching with Regular Expressions

In addition to the searching facilities described above, JOVE can search for patterns using regular expressions. The handling of regular expressions in JOVE is like that of *ed(1)* or *vi(1)*, but with some notable additions. The extra metacharacters understood by JOVE are \<, \>, \| and \{. The first two of these match the beginnings and endings of words; Thus the search pattern, "\<Exec" would match all words beginning with the letters "Exec".

An \| signals the beginning of an alternative — that is, the pattern "foo\|bar" would match either "foo" or "bar". The "curly brace" is a way of introducing several sub-alternatives into a pattern. It parallels the [] construct of regular expressions, except it specifies a list of alternative words instead of just alternative characters. So the pattern "foo\{bar,baz\}bie" matches "foobarbie" or "foobazbie".

JOVE only regards metacharacters as special if the variable *match-regular-expressions* is set to "on". The ability to have JOVE ignore these characters is useful if you're editing a document about patterns and regular expressions or when a novice is learning JOVE.

Another variable that affects searching is *case-ignore-search*. If this variable is set to "on" then upper case and lower case letters are considered equal.

7. Replacement Commands

Global search-and-replace operations are not needed as often in JOVE as they are in other editors, but they are available. In addition to the simple Replace operation which is like that found in most editors, there is a Query Replace operation which asks, for each occurrence of the pattern, whether to replace it.

7.1. Global replacement

To replace every occurrence of FOO after point with BAR, you can do, e.g., "ESC R FOO<return>BAR" as the *replace-string* command is bound to the ESC R. Replacement takes place only between point and the end of the buffer so if you want to cover the whole buffer you must go to the beginning first.

7.2. Query Replace

If you want to change only some of the occurrences of FOO, not all, then the global *replace-string* is inappropriate; Instead, use, e.g., "ESC Q FOO<return>BAR", to run the command *query-replace-string*. This displays each occurrence of FOO and waits for you to say whether to replace it with a BAR. The things you can type when you are shown an occurrence of FOO are:

Space	to replace the FOO.
Rubout	to skip to the next FOO without replacing this one.
Return	to stop without doing any more replacements.
Period	to replace this FOO and then stop.
! or P	to replace all remaining FOO's without asking.
C-R or R	to enter a recursive editing level, in case the FOO needs to be edited rather than just replaced with a BAR. When you are done, exit the recursive editing level with C-X C-C and the next FOO will be displayed.
C-W	to delete the FOO, and then start editing the buffer. When you are finished editing whatever is to replace the FOO, exit the recursive editing level with C-X C-C and the next FOO will be displayed.
U	move to the last replacement and undo it.

Another alternative is using *replace-in-region* which is just like *replace-string* except it searches only within the region.

8. Commands for English Text

JOVE has many commands that work on the basic units of English text: words, sentences and paragraphs.

8.1. Word Commands

JOVE has commands for moving over or operating on words. By convention, they are all ESC commands.

ESC F	Move Forward over a word.
ESC B	Move Backward over a word.
ESC D	Kill forward to the end of a word.
ESC Rubout	Kill backward to the beginning of a word.

Notice how these commands form a group that parallels the character- based commands, C-F, C-B, C-D, and Rubout.

The commands ESC F and ESC B move forward and backward over words. They are thus analogous to Control-F and Control-B, which move over single characters. Like their Control- analogues, ESC F and ESC B move several words if given an argument. ESC F with a negative argument moves backward like ESC B, and ESC B with a negative argument moves forward. Forward motion stops right after the last letter of the word, while backward motion stops right before the first letter.

It is easy to kill a word at a time. ESC D kills the word after point. To be precise, it kills everything from point to the place ESC F would move to. Thus, if point is in the middle of a word, only the part after point is killed. If some punctuation comes after point, and before the next word, it is killed along with the word. If you wish to kill only the next word but not the punctuation, simply do ESC F to get to the end, and kill the word backwards with ESC Rubout. ESC D takes arguments just like ESC F.

ESC Rubout kills the word before point. It kills everything from point back to where ESC B would move to. If point is after the space in "FOO, BAR", then "FOO, " is killed. If you wish to kill just "FOO", then do a ESC B and a ESC D instead of a ESC Rubout.

8.2. Sentence Commands

The JOVE commands for manipulating sentences and paragraphs are mostly ESC commands, so as to resemble the word-handling commands.

ESC A	Move back to the beginning of the sentence.
ESC E	Move forward to the end of the sentence.
ESC K	Kill forward to the end of the sentence.
C-X Rubout	Kill back to the beginning of the sentence.

The commands ESC A and ESC E move to the beginning and end of the current sentence, respectively. They were chosen to resemble Control-A and Control-E, which move to the beginning and end of a line. Unlike them, ESC A and ESC E if repeated or given numeric arguments move over successive sentences. JOVE considers a sentence to end wherever there is a ".", "?", or "!" followed by the end of a line or by one or more spaces. Neither ESC A nor ESC E moves past the end of the line or spaces which delimit the sentence.

Just as C-A and C-E have a kill command, C-K, to go with them, so ESC A and ESC E have a corresponding kill command ESC K which kills from point to the end of the sentence. With minus one as an argument it kills back to the beginning of the sentence. Positive arguments serve as a repeat count.

There is a special command, C-X Rubout for killing back to the beginning of a sentence, because this is useful when you change your mind in the middle of composing text.

8.3. Paragraph Commands

The JOVE commands for handling paragraphs are

ESC [	Move back to previous paragraph beginning.

ESC]	Move forward to next paragraph end.

ESC [moves to the beginning of the current or previous paragraph, while ESC] moves to the end of the current or next paragraph. Paragraphs are delimited by lines of differing indent, or lines with text formatter commands, or blank lines. JOVE knows how to deal with most indented paragraphs correctly, although it can get confused by one- or two-line paragraphs delimited only by indentation.

8.4. Text Indentation Commands

Tab	Indent "appropriately" in a mode-dependent fashion.
LineFeed	Is the same as Return, except it copies the indent of the line you just left.
ESC M	Moves to the line's first non-blank character.

The way to request indentation is with the Tab command. Its precise effect depends on the major mode. In *Text* mode, it indents to the next tab stop. In *C* mode, it indents to the "right" position for C programs.

To move over the indentation on a line, do ESC M (*first-non-blank*). This command, given anywhere on a line, positions the cursor at the first non-blank, non-tab character on the line.

8.5. Text Filling

Auto Fill mode causes text to be *filled* (broken up into lines that fit in a specified width) automatically as you type it in. If you alter existing text so that it is no longer properly filled, JOVE can fill it again if you ask.

Entering *Auto Fill* mode is done with ESC X *auto-fill-mode*. From then on, lines are broken automatically at spaces when they get longer than the desired width. To leave *Auto Fill* mode, once again execute ESC X *auto-fill-mode*. When *Auto Fill* mode is in effect, the word **Fill** appears in the mode line.

If you edit the middle of a paragraph, it may no longer correctly be filled. To refill a paragraph, use the command ESC J (*fill-paragraph*). It causes the paragraph that point is inside to be filled. All the line breaks are removed and new ones inserted where necessary.

The maximum line width for filling is in the variable *right-margin*. Both ESC J and auto-fill make sure that no line exceeds this width. The value of *right-margin* is initially 72.

Normally ESC J figures out the indent of the paragraph and uses that same indent when filling. If you want to change the indent of a paragraph you set *left-margin* to the new position and type C-U ESC J. *fill-paragraph*, when supplied a numeric argument, uses the value of *left-margin*.

If you know where you want to set the right margin but you don't know the actual value, move to where you want to set the value and use the *right-margin-here* command. *left-margin-here* does the same for the *left-margin* variable.

8.6. Case Conversion Commands

ESC L	Convert following word to lower case.
ESC U	Convert following word to upper case.
ESC C	Capitalize the following word.

The word conversion commands are most useful. ESC L converts the word after point to lower case, moving past it. Thus, successive ESC L's convert successive words. ESC U converts to all capitals instead, while ESC C puts the first letter of the word into upper case and the rest into lower case. All these commands convert several words at once if given an argument. They are especially convenient for converting a large amount of text from all upper case to mixed case, because you can move through the test using ESC L, ESC U or ESC C on each word as appropriate.

When given a negative argument, the word case conversion commands apply to the appropriate number of words before point, but do not move point. This is convenient when you have just typed a word in the wrong case. You can give the case conversion command and continue typing.

If a word case conversion command is given in the middle of a word, it applies only to the part of the word which follows the cursor, treating it as a whole word.

The other case conversion functions are *case-region-upper* and *case-region-lower*, which convert everything between point and mark to the specified case. Point and mark remain unchanged.

8.7. Commands for Fixing Typos

In this section we describe the commands that are especially useful for the times when you catch a mistake on your text after you have made it, or change your mind while composing text on line.

Rubout	Delete last character.
ESC Rubout	Kill last word.
C-X Rubout	Kill to beginning of sentence.
C-T	Transpose two characters.
C-X C-T	Transpose two lines.
ESC Minus ESC L	Convert last word to lower case.
ESC Minus ESC U	Convert last word to upper case.
ESC Minus ESC C	Convert last word to lower case with capital initial.

8.8. Killing Your Mistakes

The Rubout command is the most important correction command. When used among printing (self-inserting) characters, it can be thought of as canceling the last character typed.

When your mistake is longer than a couple of characters, it might be more convenient to use ESC Rubout or C-X Rubout. ESC Rubout kills back to the start of the last word, and C-X Rubout kills back to the start of the last sentence. C-X Rubout is particularly useful when you are thinking of what to write as you type it, in case you change your mind about phrasing. ESC Rubout and C-X Rubout save the killed text for C-Y and ESC Y to retrieve.

ESC Rubout is often useful even when you have typed only a few characters wrong, if you know you are confused in your typing and aren't sure what you typed. At such a time, you cannot correct with Rubout except by looking at the screen to see what you did. It requires less thought to kill the whole word and start over again, especially if the system is heavily loaded.

If you were typing a command or command parameters, C-G will abort the command with no further processing.

8.9. Transposition

The common error of transposing two characters can be fixed with the C-T (*transpose-characters*) command. Normally, C-T transposes the two characters on either side of the cursor and moves the cursor forward one character. Repeating the command several times "drags" a character to the right. (Remember that *point* is considered to be between two characters, even though the visible cursor in your terminal is on only one of them.) When given at the end of a line, rather than switching the last character of the line with the line separator, which would be useless, C-T transposes the last two characters on the line. So, if you catch your transposition error right away, you can fix it with just a C-T. If you don't catch it so fast, you must move the cursor back to between the two characters.

To transpose two lines, use the C-X C-T (*transpose-lines*) command. The line containing the cursor is exchanged with the line above it; the cursor is left at the beginning of the line following its original position.

8.10. Checking and Correcting Spelling

When you write a paper, you should correct its spelling at some point close to finishing it. To correct the entire buffer, do ESC X *spell-buffer*. This invokes the UNIX *spell* program, which prints a list of all the misspelled words. JOVE catches the list and places it in a JOVE buffer called **Spell**. You are given an

opportunity to delete from that buffer any words that aren't really errors; then JOVE looks up each misspelled word and remembers where it is in the buffer being corrected. Then you can go forward to each misspelled word with C-X C-N (*next-error*) and backward with C-X C-P (*previous-error*). See the section entitled *Error Message Parsing*.

9. File Handling

The basic unit of stored data is the file. Each program, each paper, lives usually in its own file. To edit a program or paper, the editor must be told the name of the file that contains it. This is called *visiting* a file. To make your changes to the file permanent on disk, you must *save* the file.

9.1. Visiting Files

C-X C-V	Visit a file.
C-X C-R	Same as C-X C-V.
C-X C-S	Save the visited file.
ESC ~	Tell JOVE to forget that the buffer has been changed.

Visiting a file means copying its contents into JOVE where you can edit them. JOVE remembers the name of the file you visited. Unless you use the multiple buffer feature of JOVE, you can only be visiting one file at a time. The name of the current selected buffer is visible in the mode line.

The changes you make with JOVE are made in a copy inside JOVE. The file itself is not changed. The changed text is not permanent until you *save* it in a file. The first time you change the text, an asterisk appears at the end of the mode line; this indicates that the text contains fresh changes which will be lost unless you save them.

To visit a file, use the command C-X C-V. Follow the command with the name of the file you wish to visit, terminated by a Return. You can abort the command by typing C-G, or edit the filename with many of the standard JOVE commands (e.g., C-A, C-E, C-F, ESC F, ESC Rubout). If the filename you wish to visit is similar to the filename in the mode line (the default filename), you can type C-R to insert the default and then edit it. If you do type a Return to finish the command, the new file's text appears on the screen, and its name appears in the mode line. In addition, its name becomes the new default filename.

If you wish to save the file and make your changes permanent, type C-X C-S. After the save is finished, C-X C-S prints the filename and the number of characters and lines that it wrote to the file. If there are no changes to save (no asterisk at the end of the mode line), the file is not saved; otherwise the changes saved and the asterisk at the end of the mode line will disappear.

What if you want to create a file? Just visit it. JOVE prints *(New file)* but aside from that behaves as if you had visited an existing empty file. If you make any changes and save them, the file is created. If you visit a nonexistent file unintentionally (because you typed the wrong filename), go ahead and visit the file you meant. If you don't save the unwanted file, it is not created.

If you alter one file and then visit another in the same buffer, JOVE offers to save the old one. If you answer YES, the old file is saved; if you answer NO, all the changes you have made to it since the last save are lost. You should not type ahead after a file visiting command, because your type-ahead might answer an unexpected question in a way that you would regret.

Sometimes you will change a buffer by accident. Even if you undo the effect of the change by editing, JOVE still knows that "the buffer has been changed". You can tell JOVE to pretend that there have been no changes with the ESC ~ command (*make-buffer-unmodified*). This command simply clears the "modified" flag which says that the buffer contains changes which need to be saved. Even if the buffer really *is* changed JOVE will still act as if it were not.

If JOVE is about to save a file and sees that the date of the version on disk does not match what JOVE last read or wrote, JOVE notifies you of this fact, and asks what to do, because this probably means that something is wrong. For example, somebody else may have been editing the same file. If this is so, there is a good chance that your work or his work will be lost if you don't take the proper steps. You should first find out exactly what is going on. If you determine that somebody else has modified the file, save your file

under a different filename and then DIFF the two files to merge the two sets of changes. (The "patch" command is useful for applying the results of context diffs directly). Also get in touch with the other person so that the files don't diverge any further.

9.2. How to Undo Drastic Changes to a File

If you have made several extensive changes to a file and then change your mind about them, and you haven't yet saved them, you can get rid of them by reading in the previous version of the file. You can do this with the C-X C-V command, to visit the unsaved version of the file.

9.3. Recovering from system/editor crashes

JOVE does not have *Auto Save* mode, but it does provide a way to recover your work in the event of a system or editor crash. JOVE saves information about the files you're editing every so many changes to a buffer to make recovery possible. Since a relatively small amount of information is involved it's hardly even noticeable when JOVE does this. The variable "sync-frequency" says how often to save the necessary information, and the default is every 50 changes. 50 is a very reasonable number: if you are writing a paper you will not lose more than the last 50 characters you typed, which is less than the average length of a line.

9.4. Miscellaneous File Operations

ESC X *write-file* <file><return> writes the contents of the buffer into the file <file>, and then visits that file. It can be thought of as a way of "changing the name" of the file you are visiting. Unlike C-X C-S, *write-file* saves even if the buffer has not been changed. C-X C-W is another way of getting this command.

ESC X *insert-file* <file><return> inserts the contents of <file> into the buffer at point, leaving point unchanged before the contents. You can also use C-X C-I to get this command.

ESC X *write-region* <file><return> writes the region (the text between point and mark) to the specified file. It does not set the visited filename. The buffer is not changed.

ESC X *append-region* <file><return> appends the region to <file>. The text is added to the end of <file>.

10. Using Multiple Buffers

When we speak of "the buffer", which contains the text you are editing, we have given the impression that there is only one. In fact, there may be many of them, each with its own body of text. At any time only one buffer can be *selected* and available for editing, but it isn't hard to switch to a different one. Each buffer individually remembers which file it is visiting, what modes are in effect, and whether there are any changes that need saving.

C-X B	Select or create a buffer.
C-X C-F	Visit a file in its own buffer.
C-X C-B	List the existing buffers.
C-X K	Kill a buffer.

Each buffer in JOVE has a single name, which normally doesn't change. A buffer's name can be any length. The name of the currently selected buffer and the name of the file visited in it are visible in the mode line when you are at top level. A newly started JOVE has only one buffer, named **Main**, unless you specified files to edit in the shell command that started JOVE.

10.1. Creating and Selecting Buffers

To create a new buffer, you need only think of a name for it (say, FOO) and then do C-X B FOO<return>, which is the command C-X B (*select-buffer*) followed by the name. This makes a new, empty buffer (if one by that name didn't previously exist) and selects it for editing. The new buffer is not visiting any file, so if you try to save it you will be asked for the filename to use. Each buffer has its own

major mode; the new buffer's major mode is *Text* mode by default.

To return to buffer FOO later after having switched to another, the same command C-X B FOO<return> is used, since C-X B can tell whether a buffer named FOO exists already or not. C-X B Main<return> reselects the buffer Main that JOVE started out with. Just C-X B<return> reselects the previous buffer. Repeated C-X B<return>'s alternate between the last two buffers selected.

You can also read a file into its own newly created buffer, all with one command: C-X C-F (*find-file*), followed by the filename. The name of the buffer is the last element of the file's pathname. C-F stands for "Find", because if the specified file already resides in a buffer in your JOVE, that buffer is reselected. So you need not remember whether you have brought the file in already or not. A buffer created by C-X C-F can be reselected later with C-X B or C-X C-F, whichever you find more convenient. Nonexistent files can be created with C-X C-F just as they can with C-X C-V.

10.2. Using Existing Buffers

To get a list of all the buffers that exist, do C-X C-B (*list-buffers*). Each buffer's type, name, and visited filename is printed. An asterisk before the buffer name indicates a buffer which contains changes that have not been saved. The number that appears at the beginning of a line in a C-X C-B listing is that buffer's *buffer number*. You can select a buffer by typing its number in place of its name. If a buffer with that number doesn't already exist, a new buffer is created with that number as its name.

If several buffers have modified text in them, you should save some of them with C-X C-M (*write-modified-files*). This finds all the buffers that need saving and then saves them. Saving the buffers this way is much easier and more efficient (but more dangerous) than selecting each one and typing C-X C-S. If you give C-X C-M an argument, JOVE will ask for confirmation before saving each buffer.

ESC X *rename-buffer* <new name><return> changes the name of the currently selected buffer.

ESC X *erase-buffer* <buffer name><return> erases the contents of the <buffer name> without deleting the buffer entirely.

10.3. Killing Buffers

After you use a JOVE for a while, it may fill up with buffers which you no longer need. Eventually you can reach a point where trying to create any more results in an "out of memory" or "out of lines" error. When this happens you will want to kill some buffers with the C-X K (*delete-buffer*) command. You can kill the buffer FOO by doing C-X K FOO<return>. If you type C-X K <return> JOVE will kill the previously selected buffer. If you try to kill a buffer that needs saving JOVE will ask you to confirm it.

If you need to kill several buffers, use the command *kill-some-buffers*. This prompts you with the name of each buffer and asks for confirmation before killing that buffer.

11. Controlling the Display

Since only part of a large file will fit on the screen, JOVE tries to show the part that is likely to be interesting. The display control commands allow you to see a different part of the file.

C-L	Reposition point at a specified vertical position, OR clear and redraw the screen with point in the same place.
C-V	Scroll forwards (a screen or a few lines).
ESC V	Scroll backwards.
C-Z	Scroll forward some lines.
ESC Z	Scroll backwards some lines.

The terminal screen is rarely large enough to display all of your file. If the whole buffer doesn't fit on the screen, JOVE shows a contiguous portion of it, containing *point*. It continues to show approximately the same portion until point moves outside of what is displayed; then JOVE chooses a new portion centered around the new *point*. This is JOVE's guess as to what you are most interested in seeing, but if the guess is wrong, you can use the display control commands to see a different portion. The available screen area through which you can see part of the buffer is called *the window*, and the choice of where in the buffer to

start displaying is also called *the window*. (When there is only one window, it plus the mode line and the input line take up the whole screen).

First we describe how JOVE chooses a new window position on its own. The goal is usually to place *point* half way down the window. This is controlled by the variable *scroll-step*, whose value is the number of lines above the bottom or below the top of the window that the line containing point is placed. A value of 0 (the initial value) means center *point* in the window.

The basic display control command is C-L (*redraw-display*). In its simplest form, with no argument, it tells JOVE to choose a new window position, centering point half way from the top as usual.

C-L with a positive argument chooses a new window so as to put point that many lines from the top. An argument of zero puts point on the very top line. Point does not move with respect to the text; rather, the text and point move rigidly on the screen.

If point stays on the same line, the window is first cleared and then redrawn. Thus, two C-L's in a row are guaranteed to clear the current window. ESC C-L will clear and redraw the entire screen.

The *scrolling* commands C-V, ESC V, C-Z, and ESC Z, let you move the whole display up or down a few lines. C-V (*next-page*) with an argument shows you that many more lines at the bottom of the screen, moving the text and point up together as C-L might. C-V with a negative argument shows you more lines at the top of the screen, as does ESC V (*previous-page*) with a positive argument.

To read the buffer a window at a time, use the C-V command with no argument. It takes the last line at the bottom of the window and puts it at the top, followed by nearly a whole window of lines not visible before. Point is put at the top of the window. Thus, each C-V shows the "next page of text", except for one line of overlap to provide context. To move backward, use ESC V without an argument, which moves a whole window backwards (again with a line of overlap).

C-Z and ESC Z scroll one line forward and one line backward, respectively. These are convenient for moving in units of lines without having to type a numeric argument.

11.1. Multiple Windows

JOVE allows you to split the screen into two or more *windows* and use them to display parts of different files, or different parts of the same file.

C-X 2	Divide the current window into two smaller ones.
C-X 1	Delete all windows but the current one.
C-X D	Delete current window.
C-X N	Switch to the next window.
C-X P	Switch to the previous window.
C-X O	Same as C-X P.
C-X ^	Make this window bigger.
ESC C-V	Scroll the other window.

When using *multiple window* mode, the text portion of the screen is divided into separate parts called *windows*, which can display different pieces of text. Each window can display different files, or parts of the same file. Only one of the windows is *active;* that is the window which the cursor is in. Editing normally takes place in that window alone. To edit in another window, you would give a command to move the cursor to the other window, and then edit there.

Each window displays a mode line for the buffer it's displaying. This is useful to keep track of which window corresponds with which file. In addition, the mode line serves as a separator between windows. By setting the variable *mode-line-should-standout* to "on" you can have JOVE display the mode-line in reverse video (assuming your particular terminal has the reverse video capability).

The command C-X 2 (*split-current-window*) enters multiple window mode. A new mode line appears across the middle of the screen, dividing the text display area into two halves. Both windows contain the same buffer and display the same position in it, namely where point was at the time you issued the command. The cursor moves to the second window.

To return to viewing only one window, use the command C-X 1 (*delete-other-windows*). The current window expands to fill the whole screen, and the other windows disappear until the next C-X 2. (The buffers and their contents are unaffected by any of the window operations).

While there is more than one window, you can use C-X N (*next-window*) to switch to the next window, and C-X P (*previous-window*) to switch to the previous one. If you are in the bottom window and you type C-X N, you will be placed in the top window, and the same kind of thing happens when you type C-X P in the top window, namely you will be placed in the bottom window. C-X O is the same as C-X P. It stands for "other window" because when there are only two windows, repeated use of this command will switch between the two windows.

Often you will be editing one window while using the other just for reference. Then, the command ESC C-V (*page-next-window*) is very useful. It scrolls the next window, as if you switched to the next window, typed C-V, and switched back, without your having to do all that. With a negative argument, ESC C-V will do an ESC V in the next window.

When a window splits, both halves are approximately the same size. You can redistribute the screen space between the windows with the C-X ^ (*grow-window*) command. It makes the currently selected window grow one line bigger, or as many lines as is specified with a numeric argument. Use ESC X *shrink-window* to make the current window smaller.

11.2. Multiple Windows and Multiple Buffers

Buffers can be selected independently in each window. The C-X B command selects a new buffer in whichever window contains the cursor. Other windows' buffers do not change.

You can view the same buffer in more than one window. Although the same buffer appears in both windows, they have different values of point, so you can move around in one window while the other window continues to show the same text. Then, having found one place you wish to refer to, you can go back into the other window with C-X O or C-X P to make your changes.

If you have the same buffer in both windows, you must beware of trying to visit a different file in one of the windows with C-X C-V, because if you bring a new file into this buffer, it will replaced the old file in *both* windows. To view different files in different windows, you must switch buffers in one of the windows first (with C-X B or C-X C-F, perhaps).

A convenient "combination" command for viewing something in another window is C-X 4 (*window-find*). With this command you can ask to see any specified buffer, file or tag in the other window. Follow the C-X 4 with either B and a buffer name, F and a filename, or T and a tag name. This switches to the other window and finds there what you specified. If you were previously in one-window mode, multiple-window mode is entered. C-X 4 B is similar to C-X 2 C-X B. C-X 4 F is similar to C-X 2 C-X C-F. C-X 4 T is similar to C-X 2 C-X T. The difference is one of efficiency, and also that C-X 4 works equally well if you are already using two windows.

12. Processes Under JOVE

Another feature in JOVE is its ability to interact with UNIX in a useful way. You can run other UNIX commands from JOVE and catch their output in JOVE buffers. In this chapter we will discuss the different ways to run and interact with UNIX commands.

12.1. Non-interactive UNIX commands

To run a UNIX command from JOVE just type "C-X !" followed by the name of the command terminated with Return. For example, to get a list of all the users on the system, you do:

C-X ! who<return>

Then JOVE picks a reasonable buffer in which the output from the command will be placed. E.g., "who" uses a buffer called **who**; "ps alx" uses **ps**; and "fgrep -n foo *.c" uses **fgrep**. If JOVE wants to use a buffer that already exists it first erases the old contents. If the buffer it selects holds a file, not output from a previous shell command, you must first delete that buffer with C-X K.

Once JOVE has picked a buffer it puts that buffer in a window so you can see the command's output as it is running. If there is only one window JOVE will automatically make another one. Otherwise, JOVE tries to pick the most convenient window which isn't the current one.

It's not a good idea to type anything while the command is running. There are two reasons for this:

(i) JOVE won't see the characters (thus won't execute them) until the command finishes, so you may forget what you've typed.

(ii) Although JOVE won't know what you've typed, it *will* know that you've typed something, and then it will try to be "smart" and not update the display until it's interpreted what you've typed. But, of course, JOVE won't interpret what you type until the UNIX command completes, so you're left with the uneasy feeling you get when you don't know what the hell the computer is doing*.

If you want to interrupt the command for some reason (perhaps you mistyped it, or you changed your mind) you can type C-]. Typing this inside JOVE while a process is running is the same as typing C-C when you are outside JOVE, namely the process stops in a hurry.

When the command finishes, JOVE puts you back in the window in which you started. Then it prints a message indicating whether or not the command completed successfully in its (the command's) opinion. That is, if the command had what it considers an error (or you interrupt it with C-]) JOVE will print an appropriate message.

12.2. Limitations of Non-Interactive Processes

The reason these are called non-interactive processes is that you can't type any input to them; you can't interact with them; they can't ask you questions because there is no way for you to answer. For example, you can't run a command interpreter (a shell), or *mail* or *crypt* with C-X ! because there is no way to provide it with input. Remember that JOVE (not the process in the window) is listening to your keyboard, and JOVE waits until the process dies before it looks at what you type.

C-X ! is useful for running commands that do some output and then exit. For example, it's very useful to use with the C compiler to catch compilation error messages (see Compiling C Programs), or with the *grep* commands.

12.3. Interactive Processes — Run a Shell in a Window

Some versions of JOVE† have the capability of running interactive processes. This is more useful than non-interactive processes for certain types of jobs:

(i) You can go off and do some editing while the command is running. This is useful for commands that do sporadic output and run for fairly long periods of time.

(ii) Unlike non-interactive processes, you can type input to these. In addition, you can edit what you type with the power of all the JOVE commands *before* you send the input to the process. This is a really important feature, and is especially useful for running a shell in a window.

(iii) Because you can continue with normal editing while one of the processes is running, you can create a bunch of contexts and manage them (select them, delete them, or temporarily put them aside) with JOVE's window and buffer mechanisms.

Although we may have given an image of processes being attached to *windows,* in fact they are attached to *buffers.* Therefore, once an *i-process* is running you can select another buffer into that window, or if you wish you can delete the window altogether. If you reselect that buffer later it will be up to date. That is, even though the buffer wasn't visible it was still receiving output from the process. You don't have to worry about missing anything when the buffer isn't visible.

---

*This is a bug and should be fixed, but probably won't be for a while.

† For example, the version provided with 4.3BSD.

12.4. Advantages of Running Processes in JOVE Windows.

There are several advantages to running a shell in a window. What you type isn't seen immediately by the process; instead JOVE waits until you type an entire line before passing it on to the process to read. This means that before you type <return> all of JOVE's editing capabilities are available for fixing errors on your input line. If you discover an error at the beginning of the line, rather than erasing the whole line and starting over, you can simply move to the error, correct it, move back and continue typing.

Another feature is that you have the entire history of your session in a JOVE buffer. You don't have to worry about output from a command moving past the top of the screen. If you missed some output you can move back through it with ESC V and other commands. In addition, you can save yourself retyping a command (or a similar one) by sending edited versions of previous commands, or edit the output of one command to become a list of commands to be executed ("immediate shell scripts").

12.5. Differences between Normal and I-process Buffers

JOVE behaves differently in several ways when you are in an *i-process* buffer. Most obviously, <return> does different things depending on both your position in the buffer and on the state of the process. In the normal case, when point is at the end of the buffer, Return does what you'd expect: it inserts a line-separator and then sends the line to the process. If you are somewhere else in the buffer, possibly positioned at a previous command that you want to edit, Return will place a copy of that line (with the prompt discarded if there is one) at the end of the buffer and move you there. Then you can edit the line and type Return as in the normal case. If the process has died for some reason, Return does nothing. It doesn't even insert itself. If that happens unexpectedly, you should type ESC X *list-processes*<return> to get a list of each process and its state. If your process died abnormally, *list-processes* may help you figure out why.

12.6. How to Run a Shell in a Window

Type ESC X *i-shell*<return> to start up a shell. As with C-X !, JOVE will create a buffer, called **shell–1**, and select a window for this new buffer. But unlike C-X ! you will be left in the new window. Now, the shell process is said to be attached to **shell–1**, and it is considered an *i-process* buffer.

13. Directory Handling

To save having to use absolute pathnames when you want to edit a nearby file JOVE allows you to move around the UNIX filesystem just as the c-shell does. These commands are:

cd dir	Change to the specified directory.
pushd [dir]	Like *cd*, but save the old directory on the directory stack. With no directory argument, simply exchange the top two directories on the stack and *cd* to the new top.
popd	Take the current directory off the stack and *cd* to the directory now at the top.
dirs	Display the contents of the directory stack.

The names and behavior of these commands were chosen to mimic those in the c-shell.

14. Editing C Programs

This section details the support provided by JOVE for working on C programs.

14.1. Indentation Commands

To save having to lay out C programs "by hand", JOVE has an idea of the correct indentation of a line, based on the surrounding context. When you are in C Mode, JOVE treats tabs specially — typing a tab at the beginning of a new line means "indent to the right place". Closing braces are also handled specially, and are indented to match the corresponding open brace.

14.2. Parenthesis and Brace Matching

To check that parentheses and braces match the way you think they do, turn on *Show Match* mode (ESC X show-match-mode). Then, whenever you type a close brace or parenthesis, the cursor moves momentarily to the matching opener, if it's currently visible. If it's not visible, JOVE displays the line containing the matching opener on the message line.

14.3. C Tags

Often when you are editing a C program, especially someone else's code, you see a function call and wonder what that function does. You then search for the function within the current file and if you're lucky find the definition, finally returning to the original spot when you are done. However, if are unlucky, the function turns out to be external (defined in another file) and you have to suspend the edit, *grep* for the function name in every .c that might contain it, and finally visit the appropriate file.

To avoid this diversion or the need to remember which function is defined in which file, Berkeley UNIX has a program called *ctags(1)*, which takes a set of source files and looks for function definitions, producing a file called *tags* as its output.

JOVE has a command called C-X T (*find-tag*) that prompts you for the name of a function (a *tag*), looks up the tag reference in the previously constructed tags file, then visits the file containing that tag in a new buffer, with point positioned at the definition of the function. There is another version of this command, namely *find-tag-at-point*, that uses the identifier at *point*.

So, when you've added new functions to a module, or moved some old ones around, run the *ctags* program to regenerate the *tags* file. JOVE looks in the file specified in the *tag-file* variable. The default is "./tags", that is, the tag file in the current directory. If you wish to use an alternate tag file, you use C-U C-X T, and JOVE will prompt for a file name. If you find yourself specifying the same file again and again, you can set *tag-file* to that file, and run *find-tag* with no numeric argument.

To begin an editing session looking for a particular tag, use the *-t tag* command line option to JOVE. For example, say you wanted to look at the file containing the tag *SkipChar*, you would invoke JOVE as:

% jove -t SkipChar

14.4. Compiling Your Program

You've typed in a program or altered an existing one and now you want to run it through the compiler to check for errors. To save having to suspend the edit, run the compiler, scribble down error messages, and then resume the edit, JOVE allows you to compile your code while in the editor. This is done with the C-X C-E (*compile-it*) command. If you run *compile-it* with no argument it runs the UNIX *make* program into a buffer; If you need a special command or want to pass arguments to *make*, run *compile-it* with any argument (C-U is good enough) and you will be prompted for the command to execute.

If any error messages are produced, they are treated specially by JOVE. That treatment is the subject of the next section.

14.5. Error Message Parsing and Spelling Checking

JOVE knows how to interpret the error messages from many UNIX commands; In particular, the messages from *cc*, *grep* and *lint* can be understood. After running the *compile-it* command, the *parse-errors* command is automatically executed, and any errors found are displayed in a new buffer. The files whose names are found in parsing the error messages are each brought into JOVE buffers and the point is positioned at the first error in the first file. The commands *current-error*, C-X C-N (*next-error*), and C-X C-P (*previous-error*) can be used to traverse the list of errors.

If you already have a file called *errs* containing, say, c compiler messages then you can get JOVE to interpret the messages by invoking it as:

% jove -p errs

JOVE has a special mechanism for checking the the spelling of a document; It runs the UNIX spell program into a buffer. You then delete from this buffer all those words that are not spelling errors and then JOVE runs the *parse-spelling-errors* command to yield a list of errors just as in the last section.

15. Simple Customization

15.1. Major Modes

To help with editing particular types of file, say a paper or a C program, JOVE has several *major modes.* These are as follows:

15.1.1. Text mode

This is the default major mode. Nothing special is done.

15.1.2. C mode

This mode affects the behavior of the tab and parentheses characters. Instead of just inserting the tab, JOVE determines where the text "ought" to line up for the C language and tabs to that position instead. The same thing happens with the close brace and close parenthesis; they are tabbed to the "right" place and then inserted. Using the *auto-execute-command* command, you can make JOVE enter *C Mode* whenever you edit a file whose name ends in *.c*.

15.1.3. Lisp mode

This mode is analogous to *C Mode*, but performs the indentation needed to lay out Lisp programs properly. Note also the *grind-s-expr* command that prettyprints an *s-expression* and the *kill-mode-expression* command.

15.2. Minor Modes

In addition to the major modes, JOVE has a set of minor modes. These are as follows:

15.2.1. Auto Indent

In this mode, JOVE indents each line the same way as that above it. That is, the Return key in this mode acts as the Linefeed key ordinarily does.

15.2.2. Show Match

Move the cursor momentarily to the matching opening parenthesis when a closing parenthesis is typed.

15.2.3. Auto Fill

In *Auto Fill* mode, a newline is automatically inserted when the line length exceeds the right margin. This way, you can type a whole paper without having to use the Return key.

15.2.4. Over Write

In this mode, any text typed in will replace the previous contents. (The default is for new text to be inserted and "push" the old along.) This is useful for editing an already-formatted diagram in which you want to change some things without moving other things around on the screen.

15.2.5. Word Abbrev

In this mode, every word you type is compared to a list of word abbreviations; whenever you type an abbreviation, it is replaced by the text that it abbreviates. This can save typing if a particular word or phrase must be entered many times. The abbreviations and their expansions are held in a file that looks like:

```
abbrev:phrase
```

This file can be set up in your ~/*.joverc* with the *read-word-abbrev-file* command. Then, whenever you are editing a buffer in *Word Abbrev* mode, JOVE checks for the abbreviations you've given. See also the commands *read-word-abbrev-file, write-word-abbrev-file, edit-word-abbrevs, define-global-word-abbrev, define-mode-word-abbrev,* and *bind-macro-to-word-abbrev,* and the variable *auto-case-abbrev.*

15.3. Variables

JOVE can be tailored to suit your needs by changing the values of variables. A JOVE variable can be given a value with the *set* command, and its value displayed with the *print* command.

The variables JOVE understands are listed along with the commands in the alphabetical list at the end of this document.

15.4. Key Re-binding

Many of the commands built into JOVE are not bound to specific keys. The command handler in JOVE is used to invoke these commands and is activated by the *execute-extended-command* command (ESC X). When the name of a command typed in is unambiguous, that command will be executed. Since it is very slow to have to type in the name of each command every time it is needed, JOVE makes it possible to *bind* commands to keys. When a command is *bound* to a key any future hits on that key will invoke that command. All the printing characters are initially bound to the command *self-insert.* Thus, typing any printing character causes it to be inserted into the text. Any of the existing commands can be bound to any key. (A *key* may actually be a *control character* or an *escape sequence* as explained previously under *Command Input Conventions*).

Since there are more commands than there are keys, two keys are treated as *prefix* commands. When a key bound to one of the prefix commands is typed, the next character typed is interpreted on the basis that it was preceded by one of the prefix keys. Initially ^X and ESC are the prefix keys and many of the built in commands are initially bound to these "two stroke" keys. (For historical reasons, the Escape key is often referred to as "Meta").

15.5. Keyboard Macros

Although JOVE has many powerful commands, you often find that you have a task that no individual command can do. JOVE allows you to define your own commands from sequences of existing ones "by example"; Such a sequence is termed a *macro.* The procedure is as follows: First you type the *start-remembering* command, usually bound to C-X (. Next you "perform" the commands which as they are being executed are also remembered, which will constitute the body of the macro. Then you give the *stop-remembering* command, usually bound to C-X). You now have a *keyboard macro.* To run this command sequence again, use the command *execute-keyboard-macro,* usually bound to C-X E. You may find this bothersome to type and re-type, so there is a way to bind the macro to a key. First, you must give the keyboard macro a name using the *name-keyboard-macro* command. Then the binding is made with the *bind-macro-to-key* command. We're still not finished because all this hard work will be lost if you leave JOVE. What you do is to save your macros into a file with the *write-macros-to-file* command. There is a corresponding *read-macros-from-file* command to retrieve your macros in the next editing session.

15.6. Initialization Files

Users will likely want to modify the default key bindings to their liking. Since it would be quite annoying to have to set up the bindings each time JOVE is started up, JOVE has the ability to read in a "startup" file. Whenever JOVE is started, it reads commands from the file *.joverc* in the user's home directory. These commands are read as if they were typed to the command handler (ESC X) during an edit. There can be only one command per line in the startup file. If there is a file */usr/lib/jove/joverc*, then this file will be read before the user's *.joverc* file. This can be used to set up a system-wide default startup mode for JOVE that is tailored to the needs of that system.

The *source* command can be used to read commands from a specified file at any time during an editing session, even from inside the *.joverc* file. This means that a macro can be used to change the key bindings, e.g., to enter a mode, by reading from a specified file which contains all the necessary bindings.

16. Alphabetical List of Commands and Variables

16.1. Prefix-1 (Escape)

This reads the next character and runs a command based on the character typed. If you wait for more than a second or so before typing the next character, the message "ESC" will be printed on the message line to remind you that JOVE is waiting for another character.

16.2. Prefix-2 (C-X)

This reads the next character and runs a command based on the character typed. If you wait for more than a second or so before typing another character, the message "C-X" will be printed on the message line to remind you that JOVE is waiting for another character.

16.3. Prefix-3 (Not Bound)

This reads the next character and runs a command based on the character typed. If you wait for more than a second or so before typing the next character, the character that invoked Prefix-3 will be printed on the message line to remind you that JOVE is waiting for another one.

16.4. allow-^S-and-^Q (variable)

This variable, when set, tells JOVE that your terminal does not need to use the characters C-S and C-Q for flow control, and that it is okay to bind things to them. This variable should be set depending upon what kind of terminal you have.

16.5. allow-bad-filenames (variable)

If set, this variable permits filenames to contain "bad" characters such as those from the set *&%!"`]{}. These files are harder to deal with, because the characters mean something to the shell. The default value is "off".

16.6. append-region (Not Bound)

This appends the region to a specified file. If the file does not already exist it is created.

16.7. apropos (Not Bound)

This types out all the commands, variables and macros with the specific keyword in their names. For each command and macro that contains the string, the key sequence that can be used to execute the command or macro is printed; with variables, the current value is printed. So, to find all the commands that are related to windows, you type

ESC X apropos window<Return>

16.8. auto-case-abbrev (variable)

When this variable is on (the default), word abbreviations are adjusted for case automatically. For example, if "jove" were the abbreviation for "jonathan's own version of emacs", then typing "jove" would give you "jonathan's own version of emacs", typing "Jove" would give you "Jonathan's own version of emacs", and typing "JOVE" would give you "Jonathan's Own Version of Emacs". When this variable is "off", upper and lower case are distinguished when looking for the abbreviation, i.e., in the example above, "JOVE" and "Jove" would not be expanded unless they were defined separately.

16.9. auto-execute-command (Not Bound)

This tells JOVE to execute a command automatically when a file whose name matches a specified pattern is visited. The first argument is the command you want executed and the second is a regular expression pattern that specifies the files that apply. For example, if you want to be in show-match-mode when you edit C source files (that is, files that end with ".c" or ".h") you can type

ESC X auto-execute-command show-match-mode .*.[ch]$

16.10. auto-execute-macro (Not Bound)

This is like *auto-execute-command* except you use it to execute macros automatically instead of built-in commands.

16.11. auto-fill-mode (Not Bound)

This turns on Auto Fill mode (or off if it's currently on) in the selected buffer. When JOVE is in Auto Fill mode it automatically breaks lines for you when you reach the right margin so you don't have to remember to hit Return. JOVE uses 78 as the right margin but you can change that by setting the variable *right-margin* to another value. See the *set* command to learn how to do this.

16.12. auto-indent-mode (Not Bound)

This turns on Auto Indent mode (or off if it's currently on) in the selected buffer. When JOVE is in Auto Indent mode, Return indents the new line to the same position as the line you were just on. This is useful for lining up C code (or any other language (but what else is there besides C?)). This is out of date because of the new command called *newline-and-indent* but it remains because of several "requests" on the part of, uh, enthusiastic and excitable users, that it be left as it is.

16.13. backward-character (C-B)

This moves point backward over a single character. If point is at the beginning of the line it moves to the end of the previous line.

16.14. backward-paragraph (ESC [)

This moves point backward to the beginning of the current or previous paragraph. Paragraphs are bounded by lines that begin with a Period or Tab, or by blank lines; a change in indentation may also signal a break between paragraphs, except that JOVE allows the first line of a paragraph to be indented differently from the other lines.

16.15. backward-s-expression (ESC C-B)

This moves point backward over a s-expression. It is just like *forward-s-expression* with a negative argument.

16.16. backward-sentence (ESC A)

This moves point backward to the beginning of the current or previous sentence. JOVE considers the end of a sentence to be the characters ".", "!" or "?" followed by a Return or by one or more spaces.

16.17. backward-word (ESC B)

This moves point backward to the beginning of the current or previous word.

16.18. bad-filename-extensions (variable)

This contains a list of words separated by spaces which are to be considered bad filename extensions, and so will not be counted in filename completion. The default is ".o" so if you have jove.c and jove.o in the same directory, the filename completion will *not* complain of an ambiguity because it will ignore jove.o.

16.19. beginning-of-file (ESC <)

This moves point backward to the beginning of the buffer. This sometimes prints the "Point Pushed" message. If the top of the buffer isn't on the screen JOVE will set the mark so you can go back to where you were if you want.

16.20. beginning-of-line (C-A)

This moves point to the beginning of the current line.

16.21. beginning-of-window (ESC ,)

This moves point to the beginning of the current window. The sequence "ESC ," is the same as "ESC <" (beginning of file) except without the shift key on the "<", and can thus can easily be remembered.

16.22. bind-to-key (Not Bound)

This attaches a key to an internal JOVE command so that future hits on that key invoke that command. For example, to make "C-W" erase the previous word, you type "ESC X bind-to-key kill-previous-word C-W".

16.23. bind-macro-to-key (Not Bound)

This is like *bind-to-key* except you use it to attach keys to named macros.

16.24. bind-macro-to-word-abbrev (Not Bound)

This command allows you to bind a macro to a previously defined word abbreviation. Whenever you type the abbreviation, it will first be expanded as an abbreviation, and then the macro will be executed. Note that if the macro moves around, you should set the mark first (C-@) and then exchange the point and mark last (C-X C-X).

16.25. buffer-position (Not Bound)

This displays the current file name, current line number, total number of lines, percentage of the way through the file, and the position of the cursor in the current line.

16.26. c-mode (Not Bound)

This turns on C mode in the currently selected buffer. This is one of currently four possible major modes: Fundamental, Text, C, Lisp. When in C or Lisp mode, Tab, "}", and ")" behave a little differently from usual: They are indented to the "right" place for C (or Lisp) programs. In JOVE, the "right" place is simply the way the author likes it (but I've got good taste).

16.27. case-character-capitalize (Not Bound)

This capitalizes the character after point, i.e., the character undo the cursor. If a negative argument is supplied that many characters *before* point are upper cased.

16.28. case-ignore-search (variable)

This variable, when set, tells JOVE to treat upper and lower case as the same when searching. Thus "jove" and "JOVE" would match, and "JoVe" would match either. The default value of this variable is "off".

16.29. case-region-lower (Not Bound)

This changes all the upper case letters in the region to their lower case equivalent.

16.30. case-region-upper (Not Bound)

This changes all the lower case letters in the region to their upper case equivalent.

16.31. case-word-capitalize (ESC C)

This capitalizes the current word by making the current letter upper case and making the rest of the word lower case. Point is moved to the end of the word. If point is not positioned on a word it is first moved forward to the beginning of the next word. If a negative argument is supplied that many words *before* point are capitalized. This is useful for correcting the word just typed without having to move point to the beginning of the word yourself.

16.32. case-word-lower (ESC L)

This lower-cases the current word and leaves point at the end of it. If point is in the middle of a word the rest of the word is converted. If point is not in a word it is first moved forward to the beginning of the next word. If a negative argument is supplied that many words *before* point are converted to lower case. This is useful for correcting the word just typed without having to move point to the beginning of the word yourself.

16.33. case-word-upper (ESC U)

This upper-cases the current word and leaves point at the end of it. If point is in the middle of a word the rest of the word is converted. If point is not in a word it is first moved forward to the beginning of the next word. If a negative argument is supplied that many words *before* point are converted to upper case. This is useful for correcting the word just typed without having to move point to the beginning of the word yourself.

16.34. character-to-octal-insert (Not Bound)

This inserts a Back-slash followed by the ascii value of the next character typed. For example, "C-G" inserts the string "\007".

16.35. cd (Not Bound)

This changes the current directory.

16.36. clear-and-redraw (ESC C-L)

This clears the entire screen and redraws all the windows. Use this when JOVE gets confused about what's on the screen, or when the screen gets filled with garbage characters or output from another program.

16.37. comment-format (variable)

This variable tells JOVE how to format your comments when you run the command *fill-comment*. Its format is this:

<open pattern>%!<line header>%c<line trailer>%!<close pattern>

The %!, %c, and %! must appear in the format; everything else is optional. A newline (represented by %n) may appear in the open or close patterns. %% is the representation for %. The default comment format is for C comments. See *fill-comment* for more.

16.38. compile-it (C-X C-E)

This compiles your program by running the UNIX command "make" into a buffer, and automatically parsing the error messages that are created (if any). See the *parse-errors* and *parse-special-errors* commands. To compile a C program without "make", use "C-U C-X C-E" and JOVE will prompt for a command to run instead of make. (And then the command you type will become the default command.) You can use this to parse the output from the C compiler or the "grep" or "lint" programs.

16.39. continue-process (Not Bound)

This sends SIGCONT to the current interactive process, *if* the process is currently stopped.

16.40. copy-region (ESC W)

This takes all the text in the region and copies it onto the kill ring buffer. This is just like running *kill-region* followed by the *yank* command. See the *kill-region* and *yank* commands.

16.41. current-error (Not Bound)

This moves to the current error in the list of parsed errors. See the *next-error* and *previous-error* commands for more detailed information.

16.42. date (Not Bound)

This prints the date on the message line.

16.43. define-mode-word-abbrev (Not Bound)

This defines a mode-specific abbreviation.

16.44. define-global-word-abbrev (Not Bound)

This defines a global abbreviation.

16.45. delete-blank-lines (C-X C-O)

This deletes all the blank lines around point. This is useful when you previously opened many lines with "C-O" and now wish to delete the unused ones.

16.46. delete-buffer (C-X K)

This deletes a buffer and frees up all the memory associated with it. Be careful! Once a buffer has been deleted it is gone forever. JOVE will ask you to confirm if you try to delete a buffer that needs saving. This command is useful for when JOVE runs out of space to store new buffers.

16.47. delete-macro (Not Bound)

This deletes a macro from the list of named macros. It is an error to delete the keyboard-macro. Once the macro is deleted it is gone forever. If you are about to save macros to a file and decide you don't want to save a particular one, delete it.

16.48. delete-next-character (C-D)

This deletes the character that's just after point (that is, the character under the cursor). If point is at the end of a line, the line separator is deleted and the next line is joined with the current one.

16.49. delete-other-windows (C-X 1)

This deletes all the other windows except the current one. This can be thought of as going back into One Window mode.

16.50. delete-previous-character (Rubout)

This deletes the character that's just before point (that is, the character before the cursor). If point is at the beginning of the line, the line separator is deleted and that line is joined with the previous one.

16.51. delete-white-space (ESC \)

This deletes all the Tabs and Spaces around point.

16.52. delete-current-window (C-X D)

This deletes the current window and moves point into one of the remaining ones. It is an error to try to delete the only remaining window.

16.53. describe-bindings (Not Bound)

This types out a list containing each bound key and the command that gets invoked every time that key is typed. To make a wall chart of JOVE commands, set *send-typeout-to-buffer* to "on" and JOVE will store the key bindings in a buffer which you can save to a file and then print.

16.54. describe-command (Not Bound)

This prints some info on a specified command.

16.55. describe-key (Not Bound)

This waits for you to type a key and then tells the name of the command that gets invoked every time that key is hit. Once you have the name of the command you can use the *describe-command* command to find out exactly what it does.

16.56. describe-variable (Not Bound)

This prints some info on a specified variable.

16.57. digit (ESC [0-9])

This reads a numeric argument. When you type "ESC" followed by a number, "digit" keeps reading numbers until you type some other command. Then that command is executes with the numeric argument you specified.

16.58. digit-1 (Not Bound)

This pretends you typed "ESC 1". This is useful for terminals that have keypads that send special sequences for numbers typed on the keypad as opposed to numbers typed from the keyboard. This can save having type "ESC" when you want to specify an argument.

16.59. digit-2 (Not Bound)

This pretends you typed "ESC 2". This is useful for terminals that have keypads that send special sequences for numbers typed on the keypad as opposed to numbers typed from the keyboard. This can save having type "ESC" when you want to specify an argument.

16.60. digit-3 (Not Bound)

This pretends you typed "ESC 3". This is useful for terminals that have keypads that send special sequences for numbers typed on the keypad as opposed to numbers typed from the keyboard. This can save having type "ESC" when you want to specify an argument.

16.61. digit-4 (Not Bound)

This pretends you typed "ESC 4". This is useful for terminals that have keypads that send special sequences for numbers typed on the keypad as opposed to numbers typed from the keyboard. This can save having type "ESC" when you want to specify an argument.

16.62. digit-5 (Not Bound)

This pretends you typed "ESC 5". This is useful for terminals that have keypads that send special sequences for numbers typed on the keypad as opposed to numbers typed from the keyboard. This can save having type "ESC" when you want to specify an argument.

16.63. digit-6 (Not Bound)

This pretends you typed "ESC 6". This is useful for terminals that have keypads that send special sequences for numbers typed on the keypad as opposed to numbers typed from the keyboard. This can save having type "ESC" when you want to specify an argument.

16.64. digit-7 (Not Bound)

This pretends you typed "ESC 7". This is useful for terminals that have keypads that send special sequences for numbers typed on the keypad as opposed to numbers typed from the keyboard. This can save having type "ESC" when you want to specify an argument.

16.65. digit-8 (Not Bound)

This pretends you typed "ESC 8". This is useful for terminals that have keypads that send special sequences for numbers typed on the keypad as opposed to numbers typed from the keyboard. This can

save having type "ESC" when you want to specify an argument.

16.66. digit-9 (Not Bound)

This pretends you typed "ESC 9". This is useful for terminals that have keypads that send special sequences for numbers typed on the keypad as opposed to numbers typed from the keyboard. This can save having type "ESC" when you want to specify an argument.

16.67. digit-0 (Not Bound)

This pretends you typed "ESC 0". This is useful for terminals that have keypads that send special sequences for numbers typed on the keypad as opposed to numbers typed from the keyboard. This can save having type "ESC" when you want to specify an argument.

16.68. dirs (Not Bound)

This prints out the directory stack. See the "cd", "pushd", "popd" commands for more info.

16.69. disable-biff (variable)

When this is set, JOVE disables biff when you're editing and enables it again when you get out of JOVE, or when you pause to the parent shell or push to a new shell. (This means arrival of new mail will not be immediately apparent but will not cause indiscriminate writing on the display). The default is "off".

16.70. dstop-process (Not Bound)

Send the "dsusp" character to the current process. This is the character that suspends a process on the next read from the terminal. Most people have it set to C-Y. This only works if you have the interactive process feature, and if you are in a buffer bound to a process.

16.71. edit-word-abbrevs (Not Bound)

This creates a buffer with a list of each abbreviation and the phrase it expands into, and enters a recursive edit to let you change the abbreviations or add some more. The format of this list is "abbreviation:phrase" so if you add some more you should follow that format. It's probably simplest just to copy some already existing abbreviations and edit them. When you are done you type "C-X C-C" to exit the recursive edit.

16.72. end-of-file (ESC >)

This moves point forward to the end of the buffer. This sometimes prints the "Point Pushed" message. If the end of the buffer isn't on the screen JOVE will set the mark so you can go back to where you were if you want.

16.73. end-of-line (C-E)

This moves point to the end of the current line. If the line is too long to fit on the screen JOVE will scroll the line to the left to make the end of the line visible. The line will slide back to its normal position when you move backward past the leftmost visible character or when you move off the line altogether.

16.74. end-of-window (ESC .)

This moves point to the last character in the window.

16.75. eof-process (Not Bound)

Sends EOF to the current interactive process. This only works on versions of JOVE which run under 4.2-3 BSD VAX UNIX. You can't send EOF to processes on the 2.9 BSD PDP-11 UNIX.

16.76. erase-buffer (Not Bound)

This erases the contents of the specified buffer. This is like *delete-buffer* except it only erases the contents of the buffer, not the buffer itself. If you try to erase a buffer that needs saving you will be asked to

confirm it.

16.77. error-window-size (variable)

This is the percentage of the screen to use for the error-window on the screen. When you execute *compile-it, error-window-size* percent of the screen will go to the error window. If the window already exists and is a different size, it is made to be this size. The default value is 20%.

16.78. exchange-point-and-mark (C-X C-X)

This moves point to mark and makes mark the old point. This is for quickly moving from one end of the region to another.

16.79. execute-named-command (ESC X)

This is the way to execute a command that isn't bound to any key. When you are prompted with ": " you can type the name of the command. You don't have to type the entire name. Once the command is unambiguous you can type Space and JOVE will fill in the rest for you. If you are not sure of the name of the command, type "?" and JOVE will print a list of all the commands that you could possibly match given what you've already typed. If you don't have any idea what the command's name is but you know it has something to do with windows (for example), you can do "ESC X apropos window" and JOVE will print a list of all the commands that are related to windows. If you find yourself constantly executing the same commands this way you probably want to bind them to keys so that you can execute them more quickly. See the *bind-to-key* command.

16.80. execute-keyboard-macro (C-X E)

This executes the keyboard macro. If you supply a numeric argument the macro is executed that many times.

16.81. execute-macro (Not Bound)

This executes a specified macro. If you supply a numeric argument the macro is executed that many times.

16.82. exit-jove (C-X C-C)

This exits JOVE. If any buffers need saving JOVE will print a warning message and ask for confirmation. If you leave without saving your buffers all your work will be lost. If you made a mistake and really do want to exit then you can. If you are in a recursive editing level *exit-jove* will return you from that.

16.83. file-creation-mode (variable)

This variable has an octal value. It contains the mode (see *chmod(1)*) with which files should be created. This mode gets modified by your current umask setting (see *umask(1)*). The default value is usually *0666* or *0644*.

16.84. files-should-end-with-newline (variable)

This variable indicates that all files should always have a newline at the end. This is often necessary for line printers and the like. When set, if JOVE is writing a file whose last character is not a newline, it will add one automatically.

16.85. fill-comment (Not Bound)

This command fills in your C comments to make them pretty and readable. This filling is done according the variable *comment-format.*

```
/*
 * the default format makes comments like this.
 */
```

This can be changed by changing the format variable. Other languages may be supported by changing the

format variable appropriately. The formatter looks backwards from dot for an open comment symbol. If found, all indentation is done relative the position of the first character of the open symbol. If there is a matching close symbol, the entire comment is formatted. If not, the region between dot and the open symbol is reformatted.

16.86. fill-paragraph (ESC J)

This rearranges words between lines so that all the lines in the current paragraph extend as close to the right margin as possible, ensuring that none of the lines will be greater than the right margin. The default value for *right-margin* is 78, but can be changed with the *set* and *right-margin-here* commands. JOVE has a complicated algorithm for determining the beginning and end of the paragraph. In the normal case JOVE will give all the lines the same indent as they currently have, but if you wish to force a new indent you can supply a numeric argument to *fill-paragraph* (e.g., by typing C-U ESC J) and JOVE will indent each line to the column specified by the *left-margin* variable. See also the *left-margin* variable and *left-margin-here* command.

16.87. fill-region (Not Bound)

This is like *fill-paragraph*, except it operates on a region instead of just a paragraph.

16.88. filter-region (Not Bound)

This sends the text in the region to a UNIX command, and replaces the region with the output from that command. For example, if you are lazy and don't like to take the time to write properly indented C code, you can put the region around your C file and *filter-region* it through *cb*, the UNIX C beautifier. If you have a file that contains a bunch of lines that need to be sorted you can do that from inside JOVE too. by filtering the region through the *sort* UNIX command. Before output from the command replaces the region JOVE stores the old text in the kill ring, so if you are unhappy with the results you can easily get back the old text with "C-Y".

16.89. find-file (C-X C-F)

This visits a file into its own buffer and then selects that buffer. If you've already visited this file in another buffer, that buffer is selected. If the file doesn't yet exist, JOVE will print "(New file)" so that you know.

16.90. find-tag (C-X T)

This finds the file that contains the specified tag. JOVE looks up tags by default in the "tags" file in the current directory. You can change the default tag name by setting the *tag-file* variable to another name. If you specify a numeric argument to this command, you will be prompted for a tag file. This is a good way to specify another tag file without changing the default. If the tag cannot be found the error is reported and point stays where it is.

16.91. find-tag-at-point (Not Bound)

This finds the file that contains the tag that point is currently on. See *find-tag*.

16.92. first-non-blank (ESC M)

This moves point back to the indent of the current line.

16.93. forward-character (C-F)

This moves forward over a single character. If point is at the end of the line it moves to the beginning of the next one.

16.94. forward-paragraph (ESC])

This moves point forward to the end of the current or next paragraph. Paragraphs are bounded by lines that begin with a Period or Tab, or by blank lines; a change in indentation may also signal a break between paragraphs, except that JOVE allows the first line of a paragraph to be indented differently from the other lines.

16.95. forward-s-expression (ESC C-F)

This moves point forward over a s-expression. If the first significant character after point is "(", this moves past the matching ")". If the character begins an identifier, this moves just past it. This is mode dependent, so this will move over atoms in LISP mode and C identifiers in C mode. JOVE also matches "{".

16.96. forward-sentence (ESC E)

This moves point forward to the end of the current or next sentence. JOVE considers the end of a sentence to be the characters ".", "!" or "?" followed by a Return, or one or more spaces.

16.97. forward-word (ESC F)

This moves point forward to the end of the current or next word.

16.98. fundamental-mode (Not Bound)

This sets the major mode to Fundamental. This affects what JOVE considers as characters that make up words. For instance, Single-quote is not part of a word in Fundamental mode, but is in Text mode.

16.99. goto-line (ESC G)

If a numeric argument is supplied point moves to the beginning of that line. If no argument is supplied, point remains where it is. This is so you don't lose your place unintentionally, by accidentally hitting the "G" instead of "F".

16.100. grind-s-expr (Not Bound)

When point is positioned on a "(", this re-indents that LISP expression.

16.101. grow-window (C-X ^)

This makes the current window one line bigger. This only works when there is more than one window and provided there is room to change the size.

16.102. paren-flash () }])

This handles the C mode curly brace indentation, the Lisp mode paren indentation, and the Show Match mode paren/curly brace/square bracket flashing.

16.103. handle-tab (Tab)

This handles indenting to the "right" place in C and Lisp mode, and just inserts itself in Text mode.

16.104. i-search-forward (Not Bound)

Incremental search. Like search-forward except that instead of prompting for a string and searching for that string all at once, it accepts the string one character at a time. After each character you type as part of the search string, it searches for the entire string so far. When you like what it found, type the Return key to finish the search. You can take back a character with Rubout and the search will back up to the position before that character was typed. C-G aborts the search.

16.105. i-search-reverse (Not Bound)

Incremental search. Like search-reverse except that instead of prompting for a string and searching for that string all at once, it accepts the string one character at a time. After each character you type as part of the search string, it searches for the entire string so far. When you like what it found, type the Return key to finish the search. You can take back a character with Rubout and the search will back up to the position before that character was typed. C-G aborts the search.

16.106. insert-file (C-X C-I)

This inserts a specified file into the current buffer at point. Point is positioned at the beginning of the inserted file.

16.107. internal-tabstop (variable)

The number of spaces JOVE should print when it displays a tab character. The default value is 8.

16.108. interrupt-process (Not Bound)

This sends the interrupt character (usually C-C) to the interactive process in the current buffer. This is only for versions of JOVE that have the interactive processes feature. This only works when you are inside a buffer that's attached to a process.

16.109. i-shell (Not Bound)

This starts up an interactive shell in a window. JOVE uses "shell-1" as the name of the buffer in which the interacting takes place. See the manual for information on how to use interactive processes.

16.110. i-shell-command (Not Bound)

This is like *shell-command* except it lets you continue with your editing while the command is running. This is really useful for long running commands with sporadic output. See the manual for information on how to use interactive processes.

16.111. kill-next-word (ESC D)

This kills the text from point to the end of the current or next word.

16.112. kill-previous-word (ESC Rubout)

This kills the text from point to the beginning of the current or previous word.

16.113. kill-process (Not Bound)

This command prompts for a buffer name or buffer number (just as select-buffer does) and then sends the process in that buffer a kill signal (9).

16.114. kill-region (C-W)

This deletes the text in the region and saves it on the kill ring. Commands that delete text but save it on the kill ring all have the word "kill" in their names. Type "C-Y" to yank back the most recent kill.

16.115. kill-s-expression (ESC C-K)

This kills the text from point to the end of the current or next s-expression.

16.116. kill-some-buffers (Not Bound)

This goes through all the existing buffers and asks whether or not to kill them. If you decide to kill a buffer, and it turns out that the buffer is modified, JOVE will offer to save it first. This is useful for when JOVE runs out of memory to store lines (this only happens on PDP-11's) and you have lots of buffers that you are no longer using.

16.117. kill-to-beginning-of-sentence (C-X Rubout)

This kills from point to the beginning of the current or previous sentence.

16.118. kill-to-end-of-line (C-K)

This kills from point to the end of the current line. When point is at the end of the line the line separator is deleted and the next line is joined with current one. If a numeric argument is supplied that many lines are killed; if the argument is negative that many lines *before* point are killed; if the argument is zero the text from point to the beginning of the line is killed.

16.119. kill-to-end-of-sentence (ESC K)

This kills from point to the end of the current or next sentence. If a negative numeric argument is supplied it kills from point to the beginning of the current or previous sentence.

16.120. left-margin (variable)

This is how far lines should be indented when auto-indent mode is on, or when the *newline-and-indent* command is run (usually by typing LineFeed). It is also used by fill-paragraph and auto-fill mode. If the value is zero (the default) then the left margin is determined from the surrounding lines.

16.121. left-margin-here (Not Bound)

This sets the *left-margin* variable to the current position of point. This is an easy way to say, "Make the left margin begin here," without having to count the number of spaces over it actually is.

16.122. lisp-mode (Not Bound)

This turns on Lisp mode. Lisp mode is one of four mutually exclusive major modes: Fundamental, Text, C, and Lisp. In Lisp mode, the characters Tab and) are treated specially, similar to the way they are treated in C mode. Also, Auto Indent mode is affected, and handled specially.

16.123. list-buffers (C-X C-B)

This types out a list containing various information about each buffer. Right now that list looks like this:

(* means the buffer needs saving)

NO	Lines	Type		Name	File
--	-----	----		----	----
1	1	File		Main	[No file]
2	1	Scratch	*	Minibuf	[No file]
3	519	File	*	commands.doc	commands.doc

The first column lists the buffer's number. When JOVE prompts for a buffer name you can either type in the full name, or you can simply type the buffer's number. The second column is the number of lines in the buffer. The third says what type of buffer. There are four types: "File", "Scratch", "Process", "I-Process". "File" is simply a buffer that holds a file; "Scratch" is for buffers that JOVE uses internally; "Process" is one that holds the output from a UNIX command; "I-Process" is one that has an interactive process attached to it. The next column contains the name of the buffer. And the last column is the name of the file that's attached to the buffer. In this case, both Minibuf and commands.doc have been changed but not yet saved. In fact Minibuf won't be saved since it's an internal JOVE buffer that I don't even care about.

16.124. list-processes (Not Bound)

This makes a list somewhat like "list-buffers" does, except its list consists of the current interactive processes. Right now the list looks like this:

Buffer	Status	Command name
shell-1	Running	i-shell
fgrep	Done	fgrep -n Buffer *.c

The first column has the name of the buffer to which the process is attached. The second has the status of the process; if a process has exited normally the status is "Done" as in fgrep; if the process exited with an error the status is "Exit N" where N is the value of the exit code; if the process was killed by some signal the status is the name of the signal that was used; otherwise the process is running. The last column is the name of the command that is being run.

16.125. mailbox (variable)

Set this to the full pathname of your mailbox. JOVE will look here to decide whether or not you have any unread mail. This defaults to /usr/spool/mail/$USER, where $USER is set to your login name.

16.126. mail-check-frequency (variable)

This is how often (in seconds) JOVE should check your mailbox for incoming mail. See also the *mailbox* and *disable-biff* variables.

16.127. make-backup-files (variable)

If this variable is set, then whenever JOVE writes out a file, it will move the previous version of the file (if there was one) to "#filename". This is often convenient if you save a file by accident. The default value of this variable is "off". *Note:* this is an optional part of JOVE, and your guru may not have it enabled, so it may not work.

16.128. make-buffer-unmodified (ESC ~)

This makes JOVE think the selected buffer hasn't been changed even if it has. Use this when you accidentally change the buffer but don't want it considered changed. Watch the mode line to see the * disappear when you use this command.

16.129. make-macro-interactive (Not Bound)

This command is meaningful only while you are defining a keyboard macro. Ordinarily, when a command in a macro definition requires a trailing text argument (file name, search string, etc.), the argument you supply becomes part of the macro definition. If you want to be able to supply a different argument each time the macro is used, then while you are defining it, you should give the make-macro-interactive command just before typing the argument which will be used during the definition process. Note: you must bind this command to a key in order to use it; you can't say ESC X make-macro-interactive.

16.130. mark-threshold (variable)

This variable contains the number of lines point may move by before the mark is set. If, in a search or something, point moves by more than this many lines, the mark is set so that you may return easily. The default value of this variable is 22 (one screenful, on most terminals).

16.131. marks-should-float (variable)

When this variable is "off", the position of a mark is remembered as a line number within the buffer and a character number within the line. If you add or delete text before the mark, it will no longer point to the text you marked originally because that text is no longer at the same line and character number. When this variable is "on", the position of a mark is adjusted to compensate for each insertion and deletion. This makes marks much more sensible to use, at the cost of slowing down insertion and deletion somewhat. The default value is "on".

16.132. match-regular-expressions (variable)

When set, JOVE will match regular expressions in search patterns. This makes special the characters ., *, [,], ^, and $, and the two-character sequences \<, \>, \{, \} and \|. See the *ed(1)* manual page. the tutorial "Advanced Editing in UNIX ", and the section above "Searching with Regular Expressions" for more information.

16.133. meta-key (variable)

You should set this variable to "on" if your terminal has a real Meta key. If your terminal has such a key, then a key sequence like ESC Y can be entered by holding down Meta and typing Y.

16.134. mode-line (variable)

The format of the mode line can be determined by setting this variable. The items in the line are specified using a printf(3) format, with the special things being marked as "%x". Digits may be used between the 'x' may be:

C	check for new mail, and displays "[New mail]" if there is any (see also the mail-check-interval and disable-biff variables)
F	the current file name, with leading path stripped
M	the current list of major and minor modes
b	the current buffer name
c	the fill character (-)
d	the current directory
e	end of string--this must be the last item in the string
f	the current file name
l	the current load average (updated automatically)
m	the buffer-modified symbol (*)
n	the current buffer number
s	space, but only if previous character is not a space
t	the current time (updated automatically)
[ ]	the square brackets printed when in a recursive edit
( )	items enclosed in %(... %) will only be printed on the bottom mode line, rather than copied when the window is split

In addition, any other character is simply copied into the mode line. Characters may be escaped with a backslash. To get a feel for all this, try typing "ESC X print mode-line" and compare the result with your current mode line.

16.135. mode-line-should-standout (variable)

If set, the mode line will be printed in reverse video, if your terminal supports it. The default for this variable is "off".

16.136. name-keyboard-macro (Not Bound)

This copies the keyboard macro and gives it a name freeing up the keyboard macro so you can define some more. Keyboard macros with their own names can be bound to keys just like built in commands can. See the *read-macros-file-file* and *write-macros-to-file* commands.

16.137. newline (Return)

This divides the current line at point moving all the text to the right of point down onto the newly created line. Point moves down to the beginning of the new line.

16.138. newline-and-backup (C-O)

This divides the current line at point moving all the text to the right of point down onto the newly created line. The difference between this and "newline" is that point does not move down to the beginning of the new line.

16.139. newline-and-indent (LineFeed)

This behaves the same was as Return does when in Auto Indent mode. This makes Auto Indent mode obsolete but it remains in the name of backward compatibility.

16.140. next-error (C-X C-N)

This moves to the next error in the list of errors that were parsed with *parse-errors* or *parse-special-errors*. In one window the list of errors is shown with the current one always at the top. In another window is the file that contains the error. Point is positioned in this window on the line where the error occurred.

16.141. next-line (C-N)

This moves down to the next line.

16.142. next-page (C-V)

This displays the next page of the buffer by taking the bottom line of the window and redrawing the window with it at the top. If there isn't another page in the buffer JOVE rings the bell. If a numeric argument is supplied the screen is scrolled up that many lines; if the argument is negative the screen is scrolled down.

16.143. next-window (C-X N)

This moves into the next window. Windows live in a circular list so when you're in the bottom window and you try to move to the next one you are moved to the top window. It is an error to use this command with only one window.

16.144. number-lines-in-window (Not Bound)

This displays the line numbers for each line in the buffer being displayed. The number isn't actually part of the text; it's just printed before the actual buffer line is. To turn this off you run the command again; it toggles.

16.145. over-write-mode (Not Bound)

This turns Over Write mode on (or off if it's currently on) in the selected buffer. When on, this mode changes the way the self-inserting characters work. Instead of inserting themselves and pushing the rest of the line over to the right, they replace or over-write the existing character. Also, Rubout replaces the character before point with a space instead of deleting it. When Over Write mode is on "OvrWt" is displayed on the mode line.

16.146. page-next-window (ESC C-V)

This displays the next page in the next window. This is exactly the same as "C-X N C-V C-X P".

16.147. paren-flash-delay (variable)

How long, in tenths of seconds, JOVE should pause on a matching parenthesis in *Show* mode. The default is 5.

16.148. parse-errors (Not Bound)

This takes the list of C compilation errors (or output from another program in the same format) in the current buffer and parses them for use with the *next-error* and *previous-error* and *current-error* commands. This is a very useful tool and helps with compiling C programs and when used in conjunction with the "grep" UNIX command very helpful in making changes to a bunch of files. This command understands

errors produced by cc, cpp, and lint; plus any other program with the same format (e.g., "grep -n"). JOVE visits each file that has an error and remembers each line that contains an error. It doesn't matter if later you insert or delete some lines in the buffers containing errors; JOVE remembers where they are regardless. *next-error* is automatically executed after one of the parse commands, so you end up at the first error.

16.149. parse-special-errors (Not Bound)

This parses errors in an unknown format. Error parsing works with regular expression search strings with \(\)'s around the the file name and the line number. So, you can use *parse-special-errors* to parse lines that are in a slightly different format by typing in your own search string. If you don't know how to use regular expressions you can't use this command.

16.150. parse-spelling-errors-in-buffer (Not Bound)

This parses a list of words in the current buffer and looks them up in another buffer that you specify. This will probably go away soon.

16.151. pause-jove (ESC S)

This stops JOVE and returns control to the parent shell. This only works for users using the C-shell, and on systems that have the job control facility. To return to JOVE you type "fg" to the C-shell.

16.152. physical-tabstop (variable)

How many spaces your terminal prints when it prints a tab character.

16.153. pop-mark (Not Bound)

This gets executed when you run *set-mark* with a numeric argument. JOVE remembers the last 16 marks and you use *pop-mark* to go backward through the ring of marks. If you execute " *pop-mark* enough times you will eventually get back to where you started.

16.154. popd (Not Bound)

This pops one entry off the directory stack. Entries are pushed with the *pushd* command. The names were stolen from the C-shell and the behavior is the same.

16.155. previous-error (C-X C-P)

This is the same as *next-error* except it goes to the previous error. See *next-error* for documentation.

16.156. previous-line (C-P)

This moves up to the previous line.

16.157. previous-page (ESC V)

This displays the previous page of the current buffer by taking the top line and redrawing the window with it at the bottom. If a numeric argument is supplied the screen is scrolled down that many lines; if the argument is negative the screen is scrolled up.

16.158. previous-window (C-X P and C-X O)

This moves into the next window. Windows live in a circular list so when you're in the top window and you try to move to the previous one you are moved to the bottom window. It is an error to use this command with only one window.

16.159. print (Not Bound)

This prints the value of a JOVE variable.

16.160. print-message (Not Bound)

This command prompts for a message, and then prints it on the bottom line where JOVE messages are printed.

16.161. process-bind-to-key (Not Bound)

This command is identical to bind-to-key, except that it only affects your bindings when you are in a buffer attached to a process. When you enter the process buffer, any keys bound with this command will automatically take their new values. When you switch to a non-process buffer, the old bindings for those keys will be restored. For example, you might want to execute

```
process-bind-to-key stop-process ^Z
process-bind-to-key interrupt-process ^C
```

Then, when you start up an interactive process and switch into that buffer, C-Z will execute stop-process and C-C will execute interrupt- process. When you switch back to a non-process buffer, C-Z will go back to executing scroll-up (or whatever you have it bound to).

16.162. process-newline (Return)

This this only gets executed when in a buffer that is attached to an interactive-process. JOVE does two different things depending on where you are when you hit Return. When you're at the end of the I-Process buffer this does what Return normally does, except it also makes the line available to the process. When point is positioned at some other position that line is copied to the end of the buffer (with the prompt stripped) and point is moved there with it, so you can then edit that line before sending it to the process. This command *must* be bound to the key you usually use to enter shell commands (Return), or else you won't be able to enter any.

16.163. process-prompt (variable)

What a prompt looks like from the i-shell and i-shell-command processes. The default is "% ", the default C-shell prompt. This is actually a regular expression search string. So you can set it to be more than one thing at once using the \| operator. For instance, for LISP hackers, the prompt can be

```
"% -> <[0-9]>: ".
```

16.164. push-shell (Not Bound)

This spawns a child shell and relinquishes control to it. This works on any version of UNIX, but this isn't as good as *pause-jove* because it takes time to start up the new shell and you get a brand new environment every time. To return to JOVE you type "C-D".

16.165. pushd (Not Bound)

This pushes a directory onto the directory stack and cd's into it. It asks for the directory name but if you don't specify one it switches the top two entries no the stack. It purposely behaves the same as C-shell's *pushd*.

16.166. pwd (Not Bound)

This prints the working directory.

16.167. quadruple-numeric-argument (C-U)

This multiplies the numeric argument by 4. So, "C-U C-F" means forward 4 characters and "C-U C-U C-N" means down 16 lines.

16.168. query-replace-string (ESC Q)

This replaces the occurrences of a specified string with a specified replacement string. When an occurrence is found point is moved to it and then JOVE asks what to do. The options are:

Space	to replace this occurrence and go on to the next one.
Period	to replace this occurrence and then stop.
Rubout	to skip this occurrence and go on to the next one.
C-R	to enter a recursive edit. This lets you temporarily suspend the replace, do some editing, and then return to continue where you left off. To continue with the Query Replace type "C-X C-C" as if you were trying to exit JOVE. Normally you would but when you are in a recursive edit all it does is exit that recursive editing level.
C-W	to delete the matched string and then enter a recursive edit.
U	to undo the last replacement.
P or !	to go ahead and replace the remaining occurrences without asking.
Return	to stop the Query Replace.

The search for occurrences starts at point and goes to the end of the buffer, so to replace in the entire buffer you must first go to the beginning.

16.169. quit-process (Not Bound)

This is the same as typing "C-\" (the Quit character) to a normal UNIX process, except it sends it to the current process in JOVE. This is only for versions of JOVE that have the interactive processes feature. This only works when you are inside a buffer that's attached to a process.

16.170. quoted-insert (C-Q)

This lets you insert characters that normally would be executed as other JOVE commands. For example, to insert "C-F" you type "C-Q C-F".

16.171. read-word-abbrev-file (Not Bound)

This reads a specified file that contains a bunch of abbreviation definitions, and makes those abbreviations available. If the selected buffer is not already in Word Abbrev mode this command puts it in that mode.

16.172. read-macros-from-file (Not Bound)

This reads the specified file that contains a bunch of macro definitions, and defines all the macros that were currently defined when the file was created. See *write-macros-to-file* to see how to save macros.

16.173. redraw-display (C-L)

This centers the line containing point in the window. If that line is already in the middle the window is first cleared and then redrawn. If a numeric argument is supplied, the line is positioned at that offset from the top of the window. For example, "ESC 0 C-L" positions the line containing point at the top of the window.

16.174. recursive-edit (Not Bound)

This enters a recursive editing level. This isn't really very useful. I don't know why it's available for public use. I think I'll delete it some day.

16.175. rename-buffer (Not Bound)

This lets you rename the current buffer.

16.176. replace-in-region (Not Bound)

This is the same as *replace-string* except that it is restricted to occurrences between Point and Mark.

16.177. replace-string (ESC R)

This replaces all occurrences of a specified string with a specified replacement string. This is just like *query-replace-string* except it replaces without asking.

16.178. right-margin (variable)

Where the right margin is for *Auto Fill* mode and the *justify-paragraph* and *justify-region* commands. The default is 78.

16.179. right-margin-here (Not Bound)

This sets the *right-margin* variable to the current position of point. This is an easy way to say, "Make the right margin begin here," without having to count the number of spaces over it actually is.

16.180. save-file (C-X C-S)

This saves the current buffer to the associated file. This makes your changes permanent so you should be sure you really want to. If the buffer has not been modified *save-file* refuses to do the save. If you really do want to write the file you can use "C-X C-W" which executes *write-file*.

16.181. scroll-down (ESC Z)

This scrolls the screen one line down. If the line containing point moves past the bottom of the window point is moved up to the center of the window. If a numeric argument is supplied that many lines are scrolled; if the argument is negative the screen is scrolled up instead.

16.182. scroll-step (variable)

How many lines should be scrolled if the *previous-line* or *next-line* commands move you off the top or bottom of the screen. You may wish to decrease this variable if you are on a slow terminal.

16.183. scroll-up (C-Z)

This scrolls the screen one line up. If the line containing point moves past the top of the window point is moved down to the center of the window. If a numeric argument is supplied that many lines are scrolled; if the argument is negative the screen is scrolled down instead.

16.184. search-exit-char (variable)

Set this to the character you want to use to exit incremental search. The default is Newline, which makes i-search compatible with normal string search.

16.185. search-forward (C-S)

This searches forward for a specified search string and positions point at the end of the string if it's found. If the string is not found point remains unchanged. This searches from point to the end of the buffer, so any matches before point will be missed.

16.186. search-reverse (C-R)

This searches backward for a specified search string and positions point at the beginning if the string if it's found. If the string is not found point remains unchanged. This searches from point to the beginning of the buffer, so any matches after point will be missed.

16.187. select-buffer (C-X B)

This selects a new or already existing buffer making it the current one. You can type either the buffer name or number. If you type in the name you need only type the name until it is unambiguous, at which point typing Escape or Space will complete it for you. If you want to create a new buffer you can type Return instead of Space, and a new empty buffer will be created.

16.188. self-insert (Most Printing Characters)

This inserts the character that invoked it into the buffer at point. Initially all but a few of the printing characters are bound to *self-insert*.

16.189. send-typeout-to-buffer (variable)

When this is set JOVE will send output that normally overwrites the screen (temporarily) to a buffer instead. This affects commands like *list-buffers, list-processes,* and other commands that use command completion. The default value is "off".

16.190. set (Not Bound)

This gives a specified variable a new value. Occasionally you'll see lines like "set this variable to that value to do this". Well, you use the *set* command to do that.

16.191. set-mark (C-@)

This sets the mark at the current position in the buffer. It prints the message "Point pushed" on the message line. It says that instead of "Mark set" because when you set the mark the previous mark is still remembered on a ring of 16 marks. So "Point pushed" means point is pushed onto the ring of marks and becomes the value of "the mark". To go through the ring of marks you type "C-U C-@", or execute the *pop-mark* command. If you type this enough times you will get back to where you started.

16.192. shell (variable)

The shell to be used with all the shell commands command. If your SHELL environment variable is set, it is used as the value of *shell;* otherwise "/bin/csh" is the default.

16.193. shell-command (C-X !)

This runs a UNIX command and places the output from that command in a buffer. JOVE creates a buffer that matches the name of the command you specify and then attaches that buffer to a window. So, when you have only one window running this command will cause JOVE to split the window and attach the new buffer to that window. Otherwise, JOVE finds the most convenient of the available windows and uses that one instead. If the buffer already exists it is first emptied, except that if it's holding a file, not some output from a previous command, JOVE prints an error message and refuses to execute the command. If you really want to execute the command you should delete that buffer (saving it first, if you like) or use *shell-command-to-buffer,* and try again.

16.194. shell-command-to-buffer (Not Bound)

This is just like *shell-command* except it lets you specify the buffer to use instead of JOVE.

16.195. shell-flags (variable)

This defines the flags that are passed to shell commands. The default is "-c". See the *shell* variable to change the default shell.

16.196. show-match-mode (Not Bound)

This turns on Show Match mode (or off if it's currently on) in the selected buffer. This changes "}" and ")" so that when they are typed the are inserted as usual, and then the cursor flashes back to the matching "{" or "(" (depending on what was typed) for about half a second, and then goes back to just after the "}" or ")" that invoked the command. This is useful for typing in complicated expressions in a program. You can

change how long the cursor sits on the matching paren by setting the "paren-flash-delay" variable in tenths of a second. If the matching "{" or "(" isn't visible nothing happens.

16.197. shrink-window (Not Bound)

This makes the current window one line shorter, if possible. Windows must be at least 2 lines high, one for the text and the other for the mode line.

16.198. source (Not Bound)

This reads a bunch of JOVE commands from a file. The format of the file is the same as that in your initialization file (your ".joverc") in your main directory. There should be one command per line and it should be as though you typed "ESC X" while in JOVE. For example, here's part of my initialization file:

```
bind-to-key i-search-reverse ^R
bind-to-key i-search-forward ^S
bind-to-key pause-jove ^[S
```

What they do is make "C-R" call the *i-search-reverse* command and "C-S" call *i-search-forward* and "ESC S" call *pause-jove*.

16.199. spell-buffer (Not Bound)

This runs the current buffer through the UNIX *spell* program and places the output in buffer "Spell". Then JOVE lets you edit the list of words, expecting you to delete the ones that you don't care about, i.e., the ones you know are spelled correctly. Then the *parse-spelling-errors-in-buffer* command comes along and finds all the misspelled words and sets things up so the error commands work.

16.200. split-current-window (C-X 2)

This splits the current window into two equal parts (providing the resulting windows would be big enough) and displays the selected buffer in both windows. Use "C-X 1" to go back to 1 window mode.

16.201. start-remembering (C-X ()

This starts remembering your key strokes in the Keyboard macro. To stop remembering you type "C-X)". Because of a bug in JOVE you can't stop remembering by typing "ESC X stop-remembering"; *stop-remembering* must be bound to "C-X)" in order to make things work correctly. To execute the remembered key strokes you type "C-X E" which runs the *execute-keyboard-macro* command. Sometimes you may want a macro to accept different input each time it runs. To see how to do this, see the *make-macro-interactive* command.

16.202. stop-process (Not Bound)

This sends a stop signal (C-Z, for most people) to the current process. It only works if you have the interactive process feature, and you are in a buffer attached to a process.

16.203. stop-remembering (C-X))

This stop the definition of the keyboard macro. Because of a bug in JOVE, this must be bound to "C-X)". Anything else will not work properly.

16.204. string-length (Not Bound)

This prints the number of characters in the string that point sits in. Strings are surrounded by double quotes. JOVE knows that "\007" is considered a single character, namely "C-G", and also knows about other common ones, like "\r" (Return) and "\n" (LineFeed). This is mostly useful only for C programmers.

16.205. suspend-jove (ESC S)

This is a synonym for *pause-jove*.

16.206. sync-frequency (variable)

The temporary files used by JOVE are forced out to disk every *sync-frequency* modifications. The default is 50, which really makes good sense. Unless your system is very unstable, you probably shouldn't fool with this.

16.207. tag-file (variable)

This the name of the file in which JOVE should look up tag definitions. The default value is "./tags".

16.208. text-mode (Not Bound)

This sets the major mode to Text. Currently the other modes are Fundamental, C and Lisp mode.

16.209. transpose-characters (C-T)

This switches the character before point with the one after point, and then moves forward one. This doesn't work at the beginning of the line, and at the end of the line it switches the two characters before point. Since point is moved forward, so that the character that was before point is still before point, you can use "C-T" to drag a character down the length of a line. This command pretty quickly becomes very useful.

16.210. transpose-lines (C-X C-T)

This switches the current line with the one above it, and then moves down one so that the line that was above point is still above point. This, like *transpose-characters*, can be used to drag a line down a page.

16.211. unbind-key (Not Bound)

Use this to unbind *any* key sequence. You can use this to unbind even a prefix command, since this command does not use "key-map completion". For example, "ESC X unbind-key ESC [" unbinds the sequence "ESC [". This is useful for "turning off" something set in the system-wide ".joverc" file.

16.212. update-time-frequency (variable)

How often the mode line is updated (and thus the time and load average, if you display them). The default is 30 seconds.

16.213. use-i/d-char (variable)

If your terminal has insert/delete character capability you can tell JOVE not to use it by setting this to "off". In my opinion it is only worth using insert/delete character at low baud rates. WARNING: if you set this to "on" when your terminal doesn't have insert/delete character capability, you will get weird (perhaps fatal) results.

16.214. version (Not Bound)

Displays the version number of this JOVE.

16.215. visible-bell (variable)

Use the terminal's visible bell instead of beeping. This is set automatically if your terminal has the capability.

16.216. visible-spaces-in-window (Not Bound)

This displays an underscore character instead of each space in the window and displays a greater-than followed by spaces for each tab in the window. The actual text in the buffer is not changed; only the screen display is affected. To turn this off you run the command again; it toggles.

16.217. visit-file (C-X C-V)

This reads a specified file into the current buffer replacing the old text. If the buffer needs saving JOVE will offer to save it for you. Sometimes you use this to start over, say if you make lots of changes and then change your mind. If that's the case you don't want JOVE to save your buffer and you answer "NO" to the question.

16.218. window-find (C-X 4)

This lets you select another buffer in another window three different ways. This waits for another character which can be one of the following:

T	Finds a tag in the other window.
F	Finds a file in the other window.
B	Selects a buffer in the other window.

This is just a convenient short hand for "C-X 2" (or "C-X O" if there are already two windows) followed by the appropriate sequence for invoking each command. With this, though, there isn't the extra overhead of having to redisplay. In addition, you don't have to decide whether to type "C-X 2" or "C-X O" since "C-X 4" does the right thing.

16.219. word-abbrev-mode (Not Bound)

This turns on Word Abbrev mode (or off if it's currently on) in the selected buffer. Word Abbrev mode lets you specify a word (an abbreviation) and a phrase with which JOVE should substitute the abbreviation. You can use this to define words to expand into long phrases, e.g., "jove" can expand into "Jonathan's Own Version of Emacs"; another common use is defining words that you often misspell in the same way, e.g., "thier" => "their" or "teh" => "the". See the information on the *auto-case-abbrev* variable.

There are two kinds of abbreviations: mode specific and global. If you define a Mode specific abbreviation in C mode, it will expand only in buffers that are in C mode. This is so you can have the same abbreviation expand to different things depending on your context. Global abbreviations expand regardless of the major mode of the buffer. The way it works is this: JOVE looks first in the mode specific table, and then in the global table. Whichever it finds it in first is the one that's used in the expansion. If it doesn't find the word it is left untouched. JOVE tries to expand words as they are typed, when you type a punctuation character or Space or Return. If you are in Auto Fill mode the expansion will be filled as if you typed it yourself.

16.220. wrap-search (variable)

If set, searches will "wrap around" the ends of the buffer instead of stopping at the bottom or top. The default is "off".

16.221. write-files-on-make (variable)

When set, all modified files will be written out before calling make when the *compile-it* command is executed. The default is "on".

16.222. write-word-abbrev-file (Not Bound)

This writes the currently defined abbreviations to a specified file. They can be read back in and automatically defined with *read-word-abbrev-file*.

16.223. write-file (C-X C-W)

This saves the current buffer to a specified file, and then makes that file the default file name for this buffer. If you specify a file that already exists you are asked to confirm over-writing it.

16.224. write-macros-to-file (Not Bound)

This writes the currently defined macros to a specified file. The macros can be read back in with *read-macros-from-file* so you can define macros and still use them in other instantiations of JOVE.

16.225. write-modified-files (C-X C-M)

This saves all the buffers that need saving. If you supply a numeric argument it asks for each buffer whether you really want to save it.

16.226. write-region (Not Bound)

This writes the text in the region to a specified file. If the file already exists you are asked to confirm over-writing it.

16.227. yank (C-Y)

This undoes the last kill command. That is, it inserts the killed text at point. When you do multiple kill commands in a row, they are merged so that yanking them back with "C–Y" yanks back all of them.

16.228. yank-pop (ESC Y)

This yanks back previous killed text. JOVE has a kill ring on which the last 10 kills are stored. *Yank* yanks a copy of the text at the front of the ring. If you want one of the last ten kills you use "ESC Y" which rotates the ring so another different entry is now at the front. You can use "ESC Y" only immediately following a "C-Y" or another "ESC Y". If you supply a negative numeric argument the ring is rotated the other way. If you use this command enough times in a row you will eventually get back to where you started. Experiment with this. It's extremely useful.

An Introduction to the Revision Control System - Revised

Walter F. Tichy

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

ABSTRACT

The Revision Control System (RCS) manages software libraries. It greatly increases programmer productivity by centralizing and cataloging changes to a software project. This document describes the benefits of using a source code control system. It then gives a tutorial introduction to the use of RCS.

Functions of RCS

The Revision Control System (RCS) manages multiple revision of text files. RCS automates the storing, retrieval, logging, identification, and merging of revisions. RCS is useful for text that is revised frequently, for example programs, documentation, graphics, papers, form letters, etc. It greatly increases programmer productivity by providing the following functions:

1. RCS stores and retrieves multiple revisions of program and other text. Thus, one can maintain one or more releases while developing the next release, with a minimum of space overhead. Changes no longer destroy the original — previous revisions remain accessible.
   a. Maintains each module as a tree of revisions.
   b. Project libraries can be organized centrally, decentralized, or any way you like.
   c. RCS works for any type of text: programs, documentation, memos, papers, graphics, VLSI layouts, form letters, etc.
2. RCS maintains a complete history of changes. Thus, one can find out what happened to a module easily and quickly, without having to compare source listings or having to track down colleagues.
   a. RCS performs automatic record keeping.
   b. RCS logs all changes automatically.
   c. RCS guarantees project continuity.
3. RCS manages multiple lines of development.
4. RCS can merge multiple lines of development. Thus, when several parallel lines of development must be consolidated into one line, the merging of changes is automatic.
5. RCS flags coding conflicts. If two or more lines of development modify the same section of code, RCS can alert programmers about overlapping changes.

6. RCS resolves access conflicts. When two or more programmers wish to modify the same revision, RCS alerts the programmers and makes sure that one change will not wipe out the other one.
7. RCS provides high-level retrieval functions. Revisions can be retrieved according to ranges of revision numbers, symbolic names, dates, authors, and states.
8. RCS provides release and configuration control. Revisions can be marked as released, stable, experimental, etc. Configurations of modules can be described simply and directly.
9. RCS performs automatic identification of modules with name, revision number, creation time, author, etc. Thus, it is always possible to determine which revisions of which modules make up a given configuration.
10. Provides high-level management visibility. Thus, it is easy to track the status of a software project.
    a. RCS provides a complete change history.
    b. RCS records who did what when to which revision of which module.
11. RCS is fully compatible with existing software development tools. RCS is unobtrusive — its interface to the file system is such that all your existing software tools can be used as before.
12. RCS' basic user interface is extremely simple. The novice only needs to learn two commands. Its more sophisticated features have been tuned towards advanced software development environments and the experienced software professional.
13. RCS simplifies software distribution if customers also maintain sources with RCS. This technique assures proper identification of versions and configurations, and tracking of customer changes. Customer changes can be merged into distributed versions locally or by the development group.
14. RCS needs little extra space for the revisions (only the differences). If intermediate revisions are deleted, the corresponding differences are compressed into the shortest possible form.

Getting Started with RCS

Suppose you have a file *f.c* that you wish to put under control of RCS. Invoke the checkin command:

ci f.c

This command creates *f.c,v*, stores *f.c* into it as revision 1.1, and deletes *f.c*. It also asks you for a description. The description should be a synopsis of the contents of the file. All later checkin commands will ask you for a log entry, which should summarize the changes that you made.

Files ending in *,v* are called RCS files ("v" stands for "versions"), the others are called working files. To get back the working file *f.c* in the previous example, use the checkout command:

co f.c

This command extracts the latest revision from *f.c,v* and writes it into *f.c*. You can now edit *f.c* and check it back in by invoking:

ci f.c

Ci increments the revision number properly. If *ci* complains with the message

ci error: no lock set by <your login>

then your system administrator has decided to create all RCS files with the locking attribute set to "strict". With strict locking, you must lock the revision during the previous checkout. Thus, your last checkout should have been

co -l f.c

Locking assures that you, and only you, can check in the next update, and avoids nasty problems if several people work on the same file. Of course, it is too late now to do the checkout with locking, because you probably modified *f.c* already, and a second checkout would overwrite your changes. Instead, invoke

rcs -l f.c

This command will lock the latest revision for you, unless somebody else got ahead of you already. If someone else has the lock, you will have to negotiate your changes with them.

If your RCS file is private, i.e., if you are the only person who is going to deposit revisions into it, strict locking is not needed and you can turn it off. If strict locking is turned off, the owner of the RCS file need not have a lock for checkin; all others still do. Turning strict locking off and on is done with the command:

rcs -U f.c and **rcs -L f.c**

You can set the locking to strict or non-strict on every RCS file.

If you do not want to clutter your working directory with RCS files, create a subdirectory called RCS in your working directory, and move all your RCS files there. RCS commands will look first into that directory to find needed files. All the commands discussed above will still work, without any change. *

To avoid the deletion of the working file during checkin (should you want to continue editing), invoke

ci -l f.c

This command checks in *f.c* as usual, but performs an additional checkout with locking. Thus, it saves you one checkout operation. There is also an option **-u** for *ci* that does a checkin followed by a checkout without locking. This is useful if you want to compile

* Pairs of RCS and working files can really be specified in 3 ways: a) both are given, b) only the working file is given, and c) only the RCS file is given. Both files may have arbitrary path prefixes; RCS commands pair them up intelligently.

the file after the checkin. Both options also update the identification markers in your file (see below).

You can give *ci* the number you want assigned to a checked in revision. Assume all your revisions were numbered 1.1, 1.2, 1.3, etc., and you would like to start release 2.

The command

ci -r2 f.c or **ci -r2.1 f.c**

assigns the number 2.1 to the new revision. From then on, *ci* will number the subsequent revisions with 2.2, 2.3, etc. The corresponding *co* commands

co -r2 f.c and **co -r2.1 f.c**

retrieve the latest revision numbered 2.x and the revision 2.1, respectively. *Co* without a revision number selects the latest revision on the "trunk", i.e., the highest revision with a number consisting of 2 fields. Numbers with more than 2 fields are needed for branches. For example, to start a branch at revision 1.3, invoke

ci -r1.3.1 f.c

This command starts a branch numbered 1 at revision 1.3 and assigns the number 1.3.1.1 to the new revision. For more information about branches, see *rcsfile(5)*.

Automatic Identification

RCS can put special strings for identification into your source and object code. To obtain such identification, place the marker

$Header$

into your text, for instance inside a comment. RCS will replace this marker with a string of the form

$Header: filename revisionnumber date time author state $

You never need to touch this string, because RCS keeps it up to date automatically. To propagate the marker into your object code, simple put it into a literal character string. In C, this is done as follows:

```
static char rcsid[ ] = "$Header$";
```

The command *ident* extracts such markers from any file, even object code. Thus, *ident* helps you to find out which revisions of which modules were used in a given program.

You may also find it useful to put the marker

Log

into your text, inside a comment. This marker accumulates the log messages that are requested during checkin. Thus, you can maintain the complete history of your file

directly inside it. There are several additional identification markers; see *co(1)* for details.

Numbering of Revisions

Revisions are organized in a tree that grows from the initial revision. The tree has a main trunk, along which the revisions are normally numbered 1.1, 1.2, 1.3, ... 2.1, 2.2, ... etc. Every revision can sprout several branches. A branch starting at revision X is assigned a revision level of X.y, and revisions on that branch have numbers X.y.z. For example, branches starting at revision 1.3 are numbered 1.3.1, 1.3.2, 1.3.3, etc., and revisions on branch 1.3.1 are numbered 1.3.1.1, 1.3.1.2, 1.3.1.3, etc. Note that revisions on branches may sprout new branches, and the numbering works analogously.

Revisions and branches may also be labeled symbolically. For instance, branch 1.3.1 could be labeled "Experimental". Revisions on a labeled branch can then be identified using the branch label as a prefix. For example, revision 1.3.1.1 would be identified as "Experimental.1". Of course, it is also possible to give a symbolic name to an individual revision. This label can then be used to identify the revision and as a prefix for branches starting with that revision. Note that labels are mapped to revision numbers. Labels start with letters and are followed by letters, digits, and underbars.

How to Combine MAKE and RCS

If your RCS files are in the same directory as your working files, you can put a default rule into your makefile. Do not use a rule of the form *.c,v.c,* because such a rule keeps a copy of every working file checked out, even those you are not working on. Instead, use this:

```
.SUFFIXES: .c,v

.c,v.o:
        co -q $*.c
        cc $(CFLAGS) -c $*.c
        rm -f $*.c

prog:   f1.o f2.o ...
        cc f1.o f2.o ... -o prog
```

This rule has the following effect. If a file *f.c* does not exist, and *f.o* is older than *f.c,v,* MAKE checks out *f.c,* compiles *f.c* into *f.o,* and then deletes *f.c.* From then on, MAKE will use *f.o* until you change *f.c,v.*

If *f.c* exists (presumably because you are working on it), the default rule *.c.o* take precedence, and *f.c* is compiled into *f.o* but not deleted.

If you keep your RCS file in the directory ./RCS, all this will not work and you have to write explicit checkout rules for every file, like

```
f1.c: RCS/f1.c,v; co -q f1.c
```

Unfortunately, these rules do not have the property of removing unneeded *.c-files.*

Printing RCS Markers

RCS markers like "$Revision: 2.3 $" etc., can be pretty-printed (i.e., without the leading keyword and the $-signs) as follows.

In nroff/troff, define the following macro:

```
.de VL
\$2
..
```

The call

```
.VL $Revision: 1.2 $
```

picks out the number (actually, the value field of any RCS marker). In all manual pages, it is recommended that you use an identification section instead of an author section, which looks something like this:

```
.SH IDENTIFICATION
Author: Walter F. Tichy,
Purdue University, West Lafayette, IN, 47907.
.sp 0
Revision Number:
.VL $Revision: 3.0 $
; Release Date:
.VL $Date: 82/11/27 11:43:39 $
.
.sp 0
Copyright © 1982 by Walter F. Tichy.
```

One could use the same trick with C-macros, but, unfortunately, these macros want commas separating the arguments. Instead, the following is the only offering:

```
char *getkeyval(s)
char *s;
{       static char keyval[100];
        sscanf(s,"%*s%s",keyval);
        return keyval;
}
```

An example of using *getkeyval()* is the following greeting message:

```
printf("Program version %s0,getkeyval("$Revision 1.2 $"));
```

There is not an option in RCS that strips off the keywords, for a good reason: If the keyword is stripped off, it becomes impossible to update the keyword value automatically.

There is not a way to suppress the keyword expansion, either. If you absolutely need a

keyword in RCS format unexpanded, piece it together from two strings in C, or imbed the null-character in nroff/troff.

Enhancements to RCS for Release 3

The major differences between release 2 and release 3 of RCS are:

- Release 3 *ci* determines whether the file to be checked in is different from the previous revision. If it is not different, *ci* asks whether to do a checkin anyway, or, if **-q** is present, *ci* suppresses the checkin. This feature avoids redundant checkins. A checkin can be forced with the new option **-f.**

  The option **-l** on release 3 *ci* now works properly: After the checkin, an implicit checkout with locking occurs. The keywords are updated. A new option, **-u,** also performs an implicit checkout, but does not lock.

  The option **-k** looks through the working file to pick up keyword values for the revision number, date, author and state, and assigns them to the checked-in revision, rather then computing them from existing locks, the clock, etc. This is useful for software distribution: Suppose a file is maintained in RCS format at several sites. If an update is sent to these sites and checked in with the **-k** option, then the original revision number, date, author, and state are preserved.

- *Co* generates full path names for RCS files during the keyword expansion. (Determining the full path causes a noticeable slowdown of *co;* this can be mitigated by checking out several files in a single command.)

  A new keyword, **$Locker$,** expands to the id of the user currently holding a lock on the revision.

- The option **-L** to *rlog* omits all files that have no locks set. The option **-R** prints only the RCS file name. Try "rlog -L -R" or "rlog -L -h".

- *Rcsdiff* (a new operation) runs diff on a checked-out file and a revision in an RCS file. This is useful for figuring out what modifications were made since the last *ci. Rcsdiff* can also run *diff* on 2 revisions in an RCS file.

- *Rcsmerge* (a new operation) merges the changes between 2 revisions in an RCS file into the checked out revision.

- *Merge,* the 3-way file merge, now has an option to print the result to *stdout.*

- Release 3 RCS no longer removes suffixes of working files. In addition, the suffix for RCS files is now ",v" instead of ".v". Thus, a working file of the form "f.c" is stored into "f.c,v".

  All you have to do is to rename your existing ".v"-files. Don't forget to add the suffix of the working file, if it was stripped off.

  Note that this change restricts the length of working file names to 12 characters (RCS detects violations reliably). The ",v" was necessary to keep *make(1)* happy.

- During the initial checkin, the RCS file inherits the read and execute permission from the working file. During subsequent checkouts, the working file inherits the read and execute permission from the RCS file. Thus, an executable file containing a shell program will still be executable after a *ci-co* cycle.

  The working file is normally generated with write permission for the owner. An exception is if locking is set to strict, and checkout is without locking. The working file is then generated without write permission, resulting in an error if one tries to edit it.

- Release 3 is portable. It has been tested on a VAX-11/780 (Unix 4.1BSD), a PDP-11/70, and a PDP-11/45 (Unix 2.8BSD), and it runs on these machines without change. Porting RCS to Berkeley Unix 4.2 is trivial by changing one macro. Also included are the modifications that were necessary to run release 2 on the BBN-C70 (BBN's C-machine), IBM 4341 with VM/UTS, M68000, Intel 86/330 with Xenix-86, Onyx with V7 Unix, VAX/VMS/Eunice 2.2. However, it has not been tested on these systems. Currently. RCS is being ported to the DEC-20.

Numerous minor problems have been fixed. RCS now dies gracefully in case the file system fills up, or if there are other read/write errors. (Gracefully in this case means that RCS files are not mutilated.) RCS operations can no longer be interrupted during the renaming of RCS files (and thus will no longer throw away RCS files if interrupted). There were some problems with nil-revision numbers and with printing of nil-strings; these have all been fixed. If stdin is not a terminal, *ci* and *rcs* now suppress the prompts for the log message and the descriptive text. Calls to *getlogin()* have been replaced with *getpwuid(getuid())*. The default for overwriting working files by *co* has been changed to not overwriting. *Co* does overwrite without asking if the file is read-only (generated by unlocking checkout, but with locking set to strict.) A serious, but extremely rare problem with the regeneration of older revisions has been fixed. The comment-leader for *.h-files* is now initially set to " * ".

Lots of fixes were necessary to make RCS portable. These include sign-extension bugs, long identifiers, conflicting structure members, and expression overflows in older C-compilers. One person reported that %02d in printf doesn't work on his USG system; a macro, DATEFORM, now exists which uses either %02d or %.2d.

Additional Information on RCS

If you want to know more about RCS, for example how to work with a tree of revisions and how to use symbolic revision numbers, read the following paper:

Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering,* IEEE, Tokyo, Sept. 1982.

Taking a look at the manual page *RCSFILE(5)* should also help to understand the revision tree permitted by RCS.

Technical Note on ICON/UXB Magnetic Tape Support

Mark Muhlestein

ICON INTERNATIONAL

December 29, 1986

This document describes certain special features of ICON/UXB support for magnetic tape. It is assumed that the reader has a basic system understanding of ICON/UXB. See also the man pages for *mt(1)* and *dd(1)*.

1. Hardware

ICON/UXB supports a variety of tape drives in order to allow for the maximum in media compatibility and cost effectiveness. Currently four drive types and their associated media are supported:

1.1. CS20 Cassette Tape Drive

This drive is normally shipped as the standard drive for ICON systems. It provides approximately 22 MB of storage on a standard 500 foot cassette, and approximately 26.5 MB on a 600 foot tape. The data is stored in a serpentine fashion with four recording tracks. Data is stored in fixed length blocks of 512 bytes, similar to the QIC formats. The media is available from ICON and large computer equipment suppliers. This drive provides an inexpensive, reliable backup capability, but it does not enjoy the media interchangeability of more standard devices.

1.2. CS50 Cassette Tape Drive

This drive is similar to the CS20, but it has nine tracks instead of four. This gives it approximately 50 MB on a 500 foot tape and 60 MB on a 600 foot tape. The recording area is narrower than the CS20 in order to accommodate the extra tracks, but the new tracks are interspersed between the four tracks used by the CS20. This allows the CS50 to read tapes created on an CS20. We have seen no problems with a CS20 reading the first four tracks of a tape written on an CS50, although the drive manufacturer does not guarantee this direction of compatibility. The CS20 and the CS50 both use the same type of cassette tape media.

1.3. MT16 9–Track ½–Inch Tape Drive

This is a streaming half inch nine track tape drive which is compatible with industry standard 1600 bpi PE format tapes. The drive can write/read blocks up to 48000 bytes long, at either 25 inches per second or 100 inches per second (streaming mode).

1.4. CR60 Quarter-Inch Cartridge Tape Drive

This drive supports both the QIC-11 and QIC-24 quarter inch cartridge standards. These cartridges store up to 60 MB. The CR60 uses a direct drive mechanism which gives excellent reliability and media interchangeability.

2. Software support under ICON/UXB

2.1. Device names

There are several entries in the **/dev** directory pertaining to tape support. The different names refer to the special characteristics and options that may be specified. Currently, only one of each type of drive may be installed on a given system. ICON/UXB uses a slightly different naming convention for tape devices than other UNIX® systems, as detailed below:

2.1.1. /dev/ct0

This device name refers to the cassette tape drive, which may be either an CS20 or an CS50. Data written to this device is organized in 512 byte blocks, and ICON/UXB buffers 128 of these blocks per physical write. If the length of the data to be written is not a multiple of 512 when the device is closed, the last block is padded with zero bytes before it is written. When the device is closed after writing, one file mark is written at the end of the data†. After closing, the tape rewinds. To avoid rewinding on close use /dev/rct0.

There are several peculiarities with the cassette tape drives. First, there are only two places data can be written: at beginning of tape (after rewinding), and at end of recorded data (after 'mt fseof'). It is not sufficient to use the 'mt fsf' command to move to the end of recorded data; 'mt fseof' must be used.

Users may notice that during a space forward operation (space backward is not supported) system operation may seem suspended. This is because the current system (by virtue of the device controller) does not support SCSI disconnect/reconnect. Another feature is that the cassette tape drives return control immediately after beginning a rewind; there is no indication of when the command completes. This is done so that the common operation of rewinding does not lock up the SCSI bus. Both these problems will be cleared up when a device controller with higher performance than the cassette tape drive manufacturer's becomes available.

2.1.2. /dev/rct0

This device is exactly like /dev/ct0 except that the rewind on close is suppressed.

2.1.3. /dev/mt0

This device refers to the half inch nine track tape drive at 1600 bpi. Note that the device is *not* treated as a filesystem file (i.e. seeks are ignored, etc.). The size of the tape blocks for *write* system calls is determined by the length passed. For reads, if the buffer length is greater than or equal to the size of the record read, the entire record is passed and the number of bytes actually read is returned. If the buffer is smaller than the block from tape, the record is truncated and the excess data will be lost; the next *read* will result in another physical I/O.

2.1.4. /dev/rmt0

This device is exactly like /dev/mt0 except that the rewind on close is suppressed.

2.1.5. /dev/hmt0

This device is exactly like /dev/rmt0 except that the tape is operated at 100 ips (high speed). This mode is especially useful for spacing operations or if the data blocks are long.

UNIX is a registered trade mark of AT&T.

† In versions of ICON/UXB prior to Release 2.16, a tape mark was not automatically written when /dev/ct0 or /dev/rct0 was closed. Users upgrading from prior releases should take note of this.

2.1.6. Other possible devices for the MT16

The major device number for the MT16 driver is (in decimal) 12, and the minor device number is encoded from the following bit string:

zzrdduuu

where *zz* is always zero, *r* is a no-rewind-on-close flag, *dd* is the density and transport speed selector, and *uuu* is the drive unit. Currently, *uuu* must be zero, as only one MT16 tape drive per system is supported. The following table shows the meaning of the different density (*dd*) values:

MT16 Density Options		
Value of "*dd*"	Tape Density	Transport Speed
00	800 bpi	25 ips
01	1600 bpi	25 ips
10	3200 bpi	25 ips
11	1600 bpi	100 ips

For example, the minor device number for an MT16 using 800 bpi tapes with no rewind-on-close would be (in binary) 00100000, or (in decimal) 32. New device entries can be made using *mknod(8)* once the major and minor device numbers have been ascertained.

2.1.7. /dev/qic24, /dev/rqic24, /dev/qic11, /dev/rqic11

/dev/qic24 refers to the MTS-1 QIC drive in QIC-24 format. It is very similar to the cassette drives in operation. /dev/rqic24 is the non-rewind on close version, and /dev/qic11 and /dev/rqic11 can be used to read and write QIC-11 format tapes. If it is necessary to read a tape when the format is unknown, try QIC-24 first, then if that does not appear to work, try QIC-11.

2.2. The TAPE environment variable

The *mt(1), tar(1), dump(8),* and *restore(8)* programs look for the special environment variable TAPE for the default device name to use. This should normally be set to the "r" versions of the device names (e.g. /dev/rmt0) because *mt* operations generally do not operate correctly if a rewind is issued after the operation. The other programs (*tar, dump* and *restore*) look at TAPE and if it begins with a "/dev/r" they will remove the "r" and use the rewinding version. It is necessary to use the "f" key-flag if it is desired to use non-rewinding devices.

If TAPE is not set, these programs default to /dev/rct0.

2.3. Options to dump(8)

The *dump* program has a new option which allows the user to directly specify a tape capacity. This is useful for cartridge and cassette media because the old version made assumptions about inter-record gaps which are incorrect for these media. The new option is "c" followed by the length in megabytes‡. For example,

dump 0cf 60 /dev/qic24 /dev/sc0g

specifies a "0" level dump, on a 60 MB tape (/dev/qic24), of the filesystem on /dev/sc0g. The default dump assumes an CS20 (21 MB). For half-inch tape, it is necessary to specify a capacity of zero in order to enable the density and length options. For example,

dump 0cfsd 0 /dev/mt0 2400 1600 /dev/sc0g

specifies a "0" level dump, on a 2400 foot tape at 1600 bpi density, of the filesystem /dev/sc0g.

‡ That is, 1,000,000 bytes, not 1024×1024 bytes.

2.4. Meaning of MTIOCGET status

Each of the devices returns different status information from the MTIOCGET *ioctl* call (see /usr/include/sys/mtio.h). Certain portions are common to all SCSI drives, however. In particular, the third byte of the status tells whether various end of media, end of file, etc., conditions have been encountered. The interpretation of this byte is as follows:

fmi0kkkk

where "f" is set when a filemark was encountered on the last command, "m" is set when end of media was encountered, "i" (meaningful for MTS-3 only) indicates a block shorter than 48112 bytes was read (this is the normal condition), "0" is a zero bit, and "kkkk" is the sense key. The meaning of the various sense keys and the other status information is not normally needed, but it can be obtained from ICON if necessary.

C O M M E N T S

ICON/UXB REFERENCE MANUAL Volume 3 **P/N 172-022-004**

Your comments and suggestions are appreciated and will help us to provide you with the very best in system and application documentation. Send your comments to the address at the bottom of this page. Users who respond will be entitled to free updates of this manual for one year.

1. How would you rate this manual for COMPLETENESS? (Please Circle)
   Excellent Poor
   5 ~~~~~~~~~~~~~ 4 ~~~~~~~~~~~~~ 3 ~~~~~~~~~~~~~ 2 ~~~~~~~~~~~~~ 1 ~~~~~~~~~~~~~ 0

2. Is there any information that you feel should be included or removed?

3. How would you rate this manual for ACCURACY? (Please Circle)
   Excellent Poor
   5 ~~~~~~~~~~~~~ 4 ~~~~~~~~~~~~~ 3 ~~~~~~~~~~~~~ 2 ~~~~~~~~~~~~~ 1 ~~~~~~~~~~~~~ 0

4. Indicate the page number and nature of any error(s) found in this manual.

5. How would you rate this manual for USABILITY? (Please Circle)
   Excellent Poor
   5 ~~~~~~~~~~~~~ 4 ~~~~~~~~~~~~~ 3 ~~~~~~~~~~~~~ 2 ~~~~~~~~~~~~~ 1 ~~~~~~~~~~~~~ 0

6. Describe any format or packaging problems you have experienced with this manual and/or binder.

7. Do you have any general comments or suggestions regarding this publication or future publications?

Your Name _____

Company _____

Address _____ Phone (___)_____

City & State _____ Zip Code _____

Job Function _____

Type of Equipment Installed: _____

Icon International, Inc. *A MEMBER OF THE SANYO GROUP* P.O. Box 340 Orem, UT 84057-0340

Printed in the U.S.A. **Rev B** **172-022-004**