Documents for UNIX

VOLUME 2

T. A. Dolotta
S. B. Olsson
A. G. Petruccelli
Editors

January 1981

Laboratory 4517
Bell Telephone Laboratories, Incorporated
Murray Hill, NJ 07974

These documents were set on an AUTOLOGIC, Inc. APS-5 phototypesetter driven by the TROFF formatter operating under the UNIX system.

ANNOTATED TABLE OF CONTENTS

NOTES: All the documents included here are supplements to the ***UNIX User's Manual*** (see G.1 below); the reader's attention is also drawn to documents G.2, G.3, and G.4.

Each document listed in Sections A through F below applies to UNIX Release 4.0, unless otherwise indicated after its title.

The number of pages in each document is given after the name(s) of its author(s).

VOLUME 1

C. DOCUMENT PREPARATION

1. NROFF/TROFF
    1. ***A TROFF Tutorial***
        B. W. Kernighan (p. 14)
        A beginner's guide to phototypesetting with TROFF.
    2. ***NROFF/TROFF User's Manual***
        J. F. Ossanna (p. 37)
        Reference manual for the UNIX text formatters.
2. Macros for NROFF/TROFF
    1. ***MM—Memorandum Macros***
        D. W. Smith and J. R. Mashey (p. 69+iv)
        Reference manual for MM, the standard BTL text-formatting macros.
    2. ***Typing Documents with MM***
        D. W. Smith and E. M. Piskorik (p. 16)
        A fold-out card that summarizes the MM macros; furnished separately.
    3. ***A Macro Package for View Graphs and Slides***
        T. A. Dolotta and D. W. Smith (p. 23)
        A guide to making visual aids with TROFF.
3. TBL and EQN
    1. ***TBL—A Program to Format Tables***
        M. E. Lesk (p. 18)
        An NROFF/TROFF preprocessor that permits easy formatting of tabular matter.
    2. ***Typesetting Mathematics—User's Guide (Second Edition)***
        B. W. Kernighan and L. L. Cherry (p. 11)
        Manual for the EQN and NEQN preprocessors for TROFF and NROFF, respectively; these preprocessors allow one to specify, in an easy-to-learn language, how to typeset complex mathematical expressions.
    3. ***A System for Typesetting Mathematics***
        B. W. Kernighan and L. L. Cherry (p. 8)
        A revision of the original EQN paper (*CACM* **18**, March 1975), describing the principles behind the design of its input language and internal structure.

D. PROGRAMMING

1. C and LINT
    1. ***The C Programming Language—Reference Manual***
        D. M. Ritchie (p. 31)
        Official statement of the syntax and semantics of C; supplemented by G.9 below.
    2. ***A Guide to the C Library for UNIX Users***
        C. D. Perez (p. 20)
        An explanation of how to use the C library.
    3. ***LINT, a C Program Checker***
        S. C. Johnson (p. 11)
        A program that checks C code for syntax errors, type violations, portability problems, and a variety of potential errors.

2. FORTRAN, RATFOR, and EFL
    1. ***A Portable FORTRAN 77 Compiler***
        S. I. Feldman and P. J. Weinberger (p. 19)
        The FORTRAN 77 language and its interfaces with the operating system.
    2. ***RATFOR—A Preprocessor for a Rational FORTRAN***
        B. W. Kernighan (p. 12)
        A preprocessor that endows FORTRAN with C-like control structures and input format.
    3. ***The Programming Language EFL***
        S. I. Feldman (p. 36)
        A general-purpose computer language intended to encourage portable programming, while making use of the good features and facilities of FORTRAN.
3. UNIX Programming
    1. ***UNIX Programming (Second Edition)***
        B. W. Kernighan and D. M. Ritchie (p. 22)
        A guide to writing programs that interface to the UNIX operating system, either directly or through the Standard I/O Library.
4. MAKE
    1. ***MAKE—A Program for Maintaining Computer Programs***
        S. I. Feldman (p. 9)
        A tool for automating the recompilation of large programs.
    2. ***An Augmented Version of MAKE***
        E. G. Bradford (p. 16)
        A discussion of how to use MAKE to its fullest advantage.
5. Debuggers
    1. ***SDB—A Symbolic Debugger***
        H. P. Katseff (p. 9)
        A debugger that allows one to examine the "core image" of an aborted program.
    2. ***A Tutorial Introduction to ADB***
        J. F. Maranzano and S. R. Bourne (p. 27)
        A guide to debugging crashed systems and programs; ADB is used mostly by system programmers.

VOLUME 2

E. SUPPORTING TOOLS AND LANGUAGES

1. LEX and YACC
   1. ***LEX—A Lexical Analyzer Generator***
      M. E. Lesk and E. Schmidt (p. 19)
      A program that generates recognizers of sets of regular expressions; each regular expression can be followed by arbitrary C code that is executed when the regular expression is found.
   2. ***YACC—Yet Another Compiler-Compiler***
      S. C. Johnson (p. 33)
      A converter from a BNF specification of a language and semantic actions written in C into a compiler for that language.
2. M4 Macro Processor
   1. ***The M4 Macro Processor***
      B. W. Kernighan and D. M. Ritchie (p. 6)
      A macro processor, also useful as a front end for languages such as C and RATFOR.
3. AWK
   1. ***AWK—A Pattern Scanning and Processing Language (Second Edition)***
      A. V. Aho, B. W. Kernighan, and P. J. Weinberger (p. 8)
      A language that makes it easy to specify many data selection and transformation operations.
4. SCCS
   1. ***Source Code Control System User's Guide***
      L. E. Bonanni and C. A. Salemi (p. 27)
      A package for controlling access and changes to (possibly multiple versions of) source programs and text files.
   2. ***Function and Use of an SCCS Interface Program***
      L. E. Bonanni and A. Guyton (p. 3)
      A discussion of how to control concurrent updates to SCCS files.
5. Calculators
   1. ***BC—An Arbitrary Precision Desk-Calculator Language***
      L. L. Cherry and R. Morris (p. 14)
      A front end for DC (see below) that provides infix notation, flow control, and built-in functions.
   2. ***DC—An Interactive Desk Calculator***
      R. Morris and L. L. Cherry (p. 8)
      An interactive desk calculator program that implements arbitrary-precision integer arithmetic.
6. Graphics
   1. ***UNIX Graphics Overview***
      A. R. Feuer (p. 7)
      An introduction to the UNIX graphics facility.
   2. ***A Tutorial Introduction to the Graphics Editor***
      A. R. Feuer (p. 17)
      A guide to making graphs, drawings, and pictures on Tektronix series 4010 terminals.

3. ***STAT—A Tool for Analyzing Data***
   A. R. Feuer and A. Guyton (p. 20)
   A collection of programs that can be interconnected via the shell to analyze statistical data and display the results in graphical form.
4. ***Administrative Information for the UNIX Graphics Package***
   R. L. Chen, D. E. Pinkston, and A. Guyton (p. 6)
   A reference guide for administrators of UNIX graphics facilities.

7. RJE and Networking
   1. ***UNIX Remote Job Entry User's Guide***
      A. L. Sabsevitz and K. A. Kelleman (p. 7)
      A guide to submitting jobs to an IBM system via the UNIX Remote Job Entry (RJE) facility.
   2. ***UNIX Remote Job Entry Administrator's Guide***
      M. J. Fitton (p. 20)
      A guide to setting up RJE on both UNIX and IBM systems, and to trouble-shooting when things go wrong.
   3. ***Release 1.0 of the UNIX Virtual Protocol Machine*** (UNIX 3.0)
      P. F. Long and C. Mee, III (p. 7)
      A description of the first version of VPM; good background reading.
   4. ***Release 2.0 of the UNIX Virtual Protocol Machine*** (UNIX 3.0)
      P. F. Long and C. Mee, III (p. 20)
      A newer release of VPM; supports bit-oriented, full-duplex protocols.

8. UUCP
   1. ***A Dial-up Network of UNIX Systems***
      D. A. Nowitz and M. E. Lesk (p. 10)
      Description of the design of a dial-up UNIX network called UUCP and used for transmission and distribution of programs and text files.
   2. ***UUCP Implementation Description***
      D. A. Nowitz (p. 15)
      A detailed description of UUCP for use by administrators of UNIX systems.

9. Printer Spooler
   1. ***The Implementation of the LP Spooling System***
      J. R. Kliegman (p. 13)
      Explanation of how the LP spooler works and how it can be used as a general-purpose spooler, as well as a line-printer spooler.
   2. ***LP Administrator's Guide***
      J. R. Kliegman (p. 12)
      A guide for those who oversee the operation of LP spoolers.

F. ADMINISTRATION, MAINTENANCE, AND IMPLEMENTATION

1. Operations and FSCK
   1. ***UNIX Operations Manual***
      A. G. Petruccelli (p. 24+ii)
      Duties of a UNIX operator.
   2. ***FSCK—The UNIX File System Check Program***
      T. J. Kowalski (p. 20)
      A guide to checking and fixing UNIX file systems.

2. Accounting and System Activity
    1. ***The UNIX Accounting System***
        H. S. McCreary and A. G. Petruccelli (p. 19)
        A guide to the use and management of the UNIX accounting system.
    2. ***The UNIX System Activity Package***
        T. W. Pao (p. 8)
        A package that reports on processor utilization, terminal activity, disk and tape I/O, swapping, system calls, etc.
3. Stand-Alone I/O
    1. ***A Stand-Alone Input/Output Library***
        S. R. Eisen (p. 11)
        A guide to the stand-alone library and the stand-alone shell (SASH).
4. ETP
    1. ***The UNIX Equipment Test Package: Operational Procedures*** (UNIX 3.0)
        A. L. Chellis and T. J. Kowalski (p. 24)
        The Equipment Test Package, a collection of UNIX hardware exercisers.
5. UNIX Internals
    1. ***UNIX Implementation***
        K. Thompson (p. 10)
        An explanation of how UNIX works; reprinted from G.5 below.
    2. ***The UNIX I/O System***
        D. M. Ritchie (p. 7)
        Guide for writers of UNIX device drivers.
    3. ***UNIX on the PDP-11/23 and 11/34 Computers*** (UNIX 3.0)
        T. J. Kowalski (p. 7)
        Description of what had to be done to UNIX to make it run on the PDP-11/23 and the PDP-11/34.
    4. ***UNIX Assembler Reference Manual***
        D. M. Ritchie (p. 12)
        Describes the UNIX PDP-11 assembler; a tool of last resort.
6. C Internals
    1. ***A Tour Through the Portable C Compiler***
        S. C. Johnson (p. 25)
        A description of how the portable C compiler works.
    2. ***A Tour Through the UNIX C Compiler***
        D. M. Ritchie (p. 15)
        A description of how the PDP-11 C compiler works.
7. Security
    1. ***On the Security of UNIX***
        D. M. Ritchie (p. 3)
        Hints on how to break UNIX and how to prevent it.
    2. ***Password Security—A Case History***
        R. Morris and K. Thompson (p. 6)
        The story of how the bad guys used to be able to break the password algorithm and why they can't now, at least not so easily.

G. RECOMMENDED READING (*not included*)

1. ***UNIX User's Manual***—Release 3.0
   T. A. Dolotta, S. B. Olsson, and A. G. Petruccelli (eds.)
   Bell Laboratories (June 1980).
   The basic document for every UNIX user.
2. ***UNIX Reference Guide***
   J. C. White (compiler) and P. V. Guidi (ed.)
   Bell Laboratories (April 1981).
   A pocket-size summary of UNIX commands, macro packages, etc.
3. ***Setting up UNIX***
   R. C. Haight, M. J. Petrella, and L. A. Wehr
   Bell Laboratories.
   Procedures for installing UNIX; must reading for anyone who wants to configure and/or generate a UNIX system. (Because this document changes with each release of UNIX, it is not included here; it is distributed with each copy of the UNIX system itself.)
4. ***Administrative Advice for UNIX***
   R. C. Haight
   Bell Laboratories.
   Hints for getting UNIX up, getting it going, and keeping it going, plus some information about hardware; must reading for UNIX system administrators. (This document is distributed just like G.3 above.)
5. ***The Bell System Technical Journal***
   Vol. 57, No. 6, Part 2 (July-August 1978).
   Special issue devoted to UNIX.
6. ***Using a Command Language as the Primary Programming Tool***
   T. A. Dolotta and J. R. Mashey
   In: Beech, D. (ed.), *Command Language Directions* (Proc. Second IFIP Working Conf. on Command Languages). Amsterdam: North Holland (1980), pp. 35-55.
   A discussion of how to get the most out of the UNIX shell.
7. ***The UNIX Programming Environment***
   B. W. Kernighan and J. R. Mashey
   COMPUTER, Vol. 14, No. 4, pp. 12-24 (April 1981); an earlier version of this paper was published in *Software—Practice & Experience*, Vol. 9, No. 1, pp. 1-15 (Jan. 1979).
   A discussion of what's good about UNIX.
8. ***Software Tools***
   B. W. Kernighan and P. J. Plauger
   Reading, MA: Addison-Wesley (1976).
   A textbook for building good software tools similar to those available in UNIX.
9. ***The C Programming Language***
   B. W. Kernighan and D. M. Ritchie
   Englewood Cliffs, NJ: Prentice-Hall (1978).
   The basic book for every C programmer; contains a tutorial and many examples.
10. ***Experiences with the UNIX Time-sharing System***
    J. Lions
    Software—Practice & Experience, Vol. 9, No. 9, pp. 701-709 (September 1979).
    An enjoyable article that tells why they like UNIX in New South Wales.

11. ***The Evolution of the UNIX Time-sharing System***
D. M. Ritchie
Proc. Symposium on Language Design and Programming Methodology, Sydney, Australia (September 1979).
Ten years later, one of the creators of UNIX looks back.

12. ***The Source Code Control System***
M. J. Rochkind
IEEE Trans. Software Eng., Vol. SE-1, No. 4, pp. 364-370 (December 1975).
The motivation for, and the underlying design of, SCCS.

LEX—A Lexical Analyzer Generator

M. E. Lesk
E. Schmidt

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial look-ahead 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 that can be translated automatically to portable Fortran. It is available on the UNIX† Time-Sharing System, 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

---

† UNIX is a trademark of Bell Laboratories.

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 the UNIX Time-Sharing System, 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.

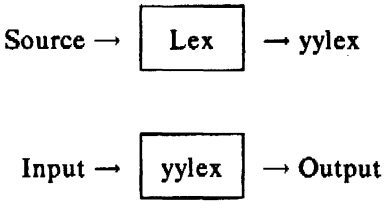


Figure 1. An Overview of Lex

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 new-line 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. 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.

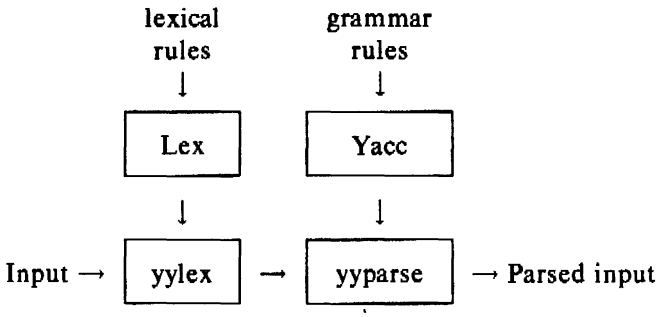


Figure 2. Lex with Yacc

Lex generates a deterministic finite automaton from the regular expressions in the source [4]. The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time taken by a Lex program to recognize and partition an input stream is proportional to the length of the input. The number of Lex rules or the complexity of the rules is not important in determining speed, unless rules which include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by Lex.

In the program written by Lex, the user's fragments (representing the ***actions*** to be performed as each regular expression is found) are gathered as cases of a switch. The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.

Lex is not limited to source which can be interpreted on the basis of one character look-ahead. 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***.

3.1 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 new-line, \t is tab, and \b is backspace. To enter \ itself, use \\. Since new-line is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, new-line and the list above is always a text character.

3.2 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.

3.3 Arbitrary character

To match almost any character, the operator character

```
.
```

is the class of all characters except new-line. Escaping into octal is possible although non-portable:

```
[\40−\176]
```

matches all printable ASCII characters, from octal 40 (blank) to octal 176 (tilde).

3.4 Optional expressions

The operator *?* indicates an optional element of an expression. Thus

```
ab?c
```

matches either ***ac*** or ***abc***.

3.5 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.

3.6 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*.

3.7 Context Sensitivity

Lex will recognize a small amount of surrounding context. The two simplest operators for this are ^ and $. If the first character of an expression is ^, the expression will only be matched at the beginning of a line (after a new-line 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 new-line). 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.

3.8 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 new-line) 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 look-ahead 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("Operator (=–) 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("Operator (=-) 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 look-ahead 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 look-ahead. Look-ahead 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 new-line. 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 new-line. 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 that begin with a blank or tab and that 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 beginning 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}             printf("real");
   ```

   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].

C programs generated by Lex on GCOS and UNIX are the same, while those on OS/370 are slightly different because the OS compiler is less powerful than the UNIX or GCOS compilers and does less at compile time.

On 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';
                      ECHO;
                      }
```

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 that recognize the need to change the environment in which the following input text is analyzed and that set a parameter to reflect the change. This may be a flag explicitly tested by the user's action code; this 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 used 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 in the same form as the character constant ***'a'***. If this interpretation is changed by providing I/O routines that translate the characters, Lex must be told about it by being given a translation table, which must be in the definitions section and must be bracketed by lines containing only ***%T***; it contains lines of the form

```
{integer} {character string}
```

which indicate the value associated with each character. Thus Figure 3 maps the lower and upper case letters together into the integers 1 through 26, new-line into 27, + and − into 28 and 29, and the digits into 30 through 39. Note the escape for new-line. 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".

```
%T
 1      Aa
 2      Bb
...
26      Zz
27      \n
28      +
29      -
30      0
31      1
...
39      9
%T
```

Figure 3. Sample Character Table

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 new-line.
^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 that 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. ACKNOWLEDGEMENTS

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, NJ (1978).

[2] B. W. Kernighan. Ratfor: A Preprocessor for a Rational Fortran, *Software−Practice & Experience* **5**:395-496 (1975).

[3] S. C. Johnson. *Yacc: Yet Another Compiler Compiler*, Bell Laboratories (1975).

[4] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search, *CACM* **18**:333-40 (1975).

[5] B. W. Kernighan, D. M. Ritchie, and K. Thompson. *QED Text Editor*, Bell Laboratories (1972).

[6] D. M. Ritchie. Private communication. See also M. E. Lesk, *The Portable C Library*, Bell Laboratories (1975).

YACC—Yet Another Compiler-Compiler

Stephen C. Johnson

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Computer program input generally has some structure; in fact, every computer program that does input can be thought of as defining an "input language" which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often are lax about checking their inputs for validity.

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher-level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.

Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

In addition to compilers for C, APL, Pascal, RATFOR, etc., Yacc has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a Fortran debugging system.

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 new-lines 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'	new-line
'\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 *end-marker*. If the tokens up to, but not including, the end-marker form a structure which matches the start symbol, the parser function returns to its caller after the end-marker is seen; it *accepts* the input. If the end-marker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the end-marker when appropriate; see section 3, below. Usually the end-marker 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 end-marker 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 *look-ahead* 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 look-ahead 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 look-ahead 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 look-ahead 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 look-ahead 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 look-ahead token. For example, in state 56 there may be an action:

```
IF      shift 34
```

which says, in state 56, if the look-ahead 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 look-ahead 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 look-ahead 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 look-ahead token is cleared by a shift, but 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 look-ahead token is the end-marker, 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 look-ahead 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 look-ahead token. The action in state 0 on *DING* is is "shift 3", so state 3 is pushed onto the stack, and the look-ahead token is cleared. State 3 becomes the current state. The next token,

DONG, is read, becoming the look-ahead token. The action in state 3 on the token *DONG* is "shift 6", so state 6 is pushed onto the stack, and the look-ahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the look-ahead, 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 look-ahead 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 end-marker is obtained, indicated by "$end" in the *y.output* file. The action in state 1 when the end-marker is seen is to accept, successfully ending the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as *DING DONG DONG*, *DING DONG*, *DING DONG DELL DELL*, etc. A few minutes spend with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

5. AMBIGUITY AND CONFLICTS

A set of grammar rules is *ambiguous* if there is some input string that can be structured in two or more different ways. For example, the grammar rule

```
expr : expr '-' expr
```

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

```
expr - expr - expr
```

the rule allows this input to be structured as either

```
( expr - expr ) - expr
```

or as

```
expr - ( expr - expr )
```

(The first is called *left association*, the second *right association*).

Yacc detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given an input such as

```
expr - expr - expr
```

When the parser has read the second expr, the input that it has seen:

```
expr - expr
```

matches the right side of the grammar rule above. The parser could *reduce* the input by applying this rule; after applying the rule; the input is reduced to *expr* (the left side of the rule). The parser would then read the final part of the input:

```
— expr
```

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, when the parser has seen

```
expr — expr
```

it could defer the immediate application of the rule, and continue reading the input until it had seen

```
expr — expr — expr
```

It could then apply the rule to the rightmost three symbols, reducing them to *expr* and leaving

```
expr — expr
```

Now the rule can be reduced once more; the effect is to take the right associative interpretation. Thus, having read

```
expr — expr
```

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a *shift/reduce conflict*. It may also happen that the parser has a choice of two legal reductions; this is called a *reduce/reduce conflict*. Note that there are never any "Shift/shift" conflicts.

When there are shift/reduce or reduce/reduce conflicts, Yacc still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a *disambiguating rule*.

Yacc invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.
2. In a reduce/reduce conflict, the default is to reduce by the *earlier* grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided whenever possible.

Conflicts may arise because of mistakes in input or logic, or because the grammar rules, while consistent, require a more complex parser than Yacc can construct. The use of actions within rules can also cause conflicts, if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate, and leads to an incorrect parser. For this reason, Yacc always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural, and produces slower parsers; thus, Yacc will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an "if-then-else" construction:

```
stat    :       IF '(' cond ')' stat
        |       IF '(' cond ')' stat ELSE stat
        ;
```

In these rules, *IF* and *ELSE* are tokens, *cond* is a nonterminal symbol describing conditional (logical) expressions, and *stat* is a nonterminal symbol describing statements. The first rule will be called the *simple-if* rule, and the second the *if-else* rule.

These two rules form an ambiguous construction, since input of the form

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be structured according to these rules in two ways:

```
IF ( C1 ) {
        IF ( C2 ) S1
        }
ELSE S2
```

or

```
IF ( C1 ) {
        IF ( C2 ) S1
        ELSE S2
        }
```

The second interpretation is the one given in most programming languages having this construct. Each *ELSE* is associated with the last preceding "un-*ELSE*'d" *IF*. In this example, consider the situation where the parser has seen

```
IF ( C1 ) IF ( C2 ) S1
```

and is looking at the *ELSE*. It can immediately reduce by the simple-if rule to get

```
IF ( C1 ) stat
```

and then read the remaining input,

```
ELSE S2
```

and reduce

```
IF ( C1 ) stat ELSE S2
```

by the if-else rule. This leads to the first of the above groupings of the input.

On the other hand, the *ELSE* may be shifted, *S2* read, and then the right hand portion of

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be reduced by the if-else rule to get

```
IF ( C1 ) stat
```

which can be reduced by the simple-if rule. This leads to the second of the above groupings of the input, which is usually desired.

Once again the parser can do two valid things — there is a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This shift/reduce conflict arises only when there is a particular current input symbol, *ELSE*, and particular inputs already seen, such as

```
IF ( C1 ) IF ( C2 ) S1
```

In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of Yacc are best understood by examining the verbose (−v) option output file. For example, the output corresponding to the above conflict state might be:

```
23: shift/reduce conflict (shift 45, reduce 18) on ELSE

state 23

        stat : IF ( cond ) stat_          (18)
        stat : IF ( cond ) stat_ELSE stat

        ELSE    shift 45
        .       reduce 18
```

The first line describes the conflict, giving the state and the input symbol. The ordinary state description follows, giving the grammar rules active in the state, and the parser actions. Recall that the underline marks the portion of the grammar rules which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

```
IF ( cond ) stat
```

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is *ELSE*, it is possible to shift into state 45. State 45 will have, as part of its description, the line

```
stat : IF ( cond ) stat ELSE_stat
```

since the *ELSE* will have been shifted in this state. Back in state 23, the alternative action, described by ".", is to be done if the input symbol is not mentioned explicitly in the above actions; thus, in this case, if the input symbol is not *ELSE*, the parser reduces by grammar rule 18:

```
stat : IF '(' cond ')' stat
```

Once again, notice that the numbers following "shift" commands refer to other states, while the numbers following "reduce" commands refer to grammar rule numbers. In the *y.output* file, the rule numbers are printed after those rules which can be reduced. In most one states, there will be at most reduce action possible in the state, and this will be the default command. The user who encounters unexpected shift/reduce conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate. In really tough cases, the user might need to know more about the behavior and construction of the parser than can be covered here. In this case, one of the theoretical references[2,3,4] might be consulted; the services of a local guru might also be appropriate.

6. PRECEDENCE

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of *precedence* levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
expr : expr OP expr
```

and

```
expr : UNARY expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the user specifies the precedence, or binding strength, of all the operators, and the associativity of the binary operators. This information is sufficient to allow Yacc to resolve the parsing conflicts in accordance with these rules, and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a Yacc keyword: %left, %right, or %nonassoc, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus,

```
%left '+' '-'
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star and slash, which are also left associative. The keyword %right is used to describe right associative operators, and the keyword %nonassoc is used to describe operators, like the operator .LT. in Fortran, that may not associate with themselves; thus,

```
A .LT. B .LT. C
```

is illegal in Fortran, and such an operator would be described with the keyword %nonassoc in Yacc. As an example of the behavior of these declarations, the description

```
%right '='
%left '+' '-'
%left '*' '/'

%%

expr    :       expr '=' expr
        |       expr '+' expr
        |       expr '-' expr
        |       expr '*' expr
        |       expr '/' expr
        |       NAME
        ;
```

might be used to structure the input

```
a = b = c*d - e - f*g
```

as follows:

```
a = ( b = ( ((c*d)-e) - (f*g) ) )
```

When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation, but different precedences. An example is unary and binary '-'; unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, %prec, changes the precedence level associated with a particular grammar rule. %prec appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, to make unary minus have the same precedence as multiplication the rules might resemble:

```
%left '+' '-'
%left '*' '/'

%%

expr    :       expr '+' expr
        |       expr '-' expr
        |       expr '*' expr
        |       expr '/' expr
        |       '-' expr    %prec '*'
        |       NAME
        ;
```

A token declared by %left, %right, and %nonassoc need not be, but may be, declared by %token as well.

The precedences and associativities are used by Yacc to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals that have them.
2. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the %prec construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.
3. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.
4. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by Yacc. This means that mistakes in the specification of precedences may disguise errors in the input grammar; it is a good idea to be sparing with precedences, and use them in an essentially "cookbook" fashion, until some experience has been gained. The *y.output* file is very useful in deciding whether the parser is actually doing what was intended.

7. ERROR HANDLING

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found; it is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser "restarted" after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, Yacc provides a simple, but reasonably general, feature. The token name "error" is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. The parser pops its stack until it enters a state where the token "error" is

legal. It then behaves as if the token "error" were the current look-ahead token, and performs the action encountered. The look-ahead 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 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 look-ahead 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 end-marker 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 look-ahead 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 "back-door" 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 end-markers 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 for other favors.

REFERENCES

[1] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ (1978).

[2] A. V. Aho and S. C. Johnson. "LR Parsing," *Comp. Surveys* **6**(2), pp. 99-124 (June 1974).

[3] A. V. Aho, S. C. Johnson, and J. D. Ullman. "Deterministic Parsing of Ambiguous Grammers," *CACM* **18**(8), pp. 441-52 (August 1975).

[4] A. V. Aho and J. D. Ullman. *Principles of Compiler Design*, Addison-Wesley, Reading, MA (1977).

[5] S. C. Johnson. "Lint, a C Program Checker," Bell Laboratories (December 1977).

[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," Bell Laboratories (March 1975).

[8] M. E. Lesk. "LEX—A Lexical Analyzer Generator," Bell Laboratories (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, new-lines, 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.

January 1981

The M4 Macro Processor

Brian W. Kernighan

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

M4 is a macro processor available on UNIX† and GCOS. Its primary use has been as a front end for Ratfor for those cases where parameterless macros are not adequately powerful. It has also been used for languages as disparate as C and Cobol. M4 is particularly suited for functional languages like Fortran, PL/I and C since macros are specified in a functional notation.

M4 provides features seldom found even in much larger macro processors, including

- arguments
- condition testing
- arithmetic capabilities
- string and substring functions
- file manipulation

This paper is a user's manual for M4.

INTRODUCTION

A macro processor is a useful way to enhance a programming language, to make it more palatable or more readable, or to tailor it to a particular application. The **#define** statement in C and the analogous **define** in Ratfor are examples of the basic facility provided by any macro processor — replacement of text by other text.

The M4 macro processor is an extension of a macro processor called M3 which was written by D. M. Ritchie for the AP-3 minicomputer; M3 was in turn based on a macro processor implemented for [1]. Readers unfamiliar with the basic ideas of macro processing may wish to read some of the discussion there.

M4 is a suitable front end for Ratfor and C, and has also been used successfully with Cobol. Besides the straightforward replacement of one string of text by another, it provides macros with arguments, conditional macro expansion, arithmetic, file manipulation, and some specialized string processing functions.

The basic operation of M4 is to copy its input to its output. As the input is read, however, each alphanumeric "token" (that is, string of letters and digits) is checked. If it is the name of a macro, then the name of the macro is replaced by its defining text, and the resulting string is pushed back onto the input to be rescanned. Macros may be called with arguments, in which case the arguments are collected and substituted into the right places in the defining text before it

† UNIX is a trademark of Bell Laboratories.

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 new-lines 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(2N+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 new-line; 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 new-line 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 new-lines 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.

REFERENCE

[1] B. W. Kernighan and P. J. Plauger. *Software Tools*, Addison-Wesley, 1976.

January 1981

AWK—A Pattern Scanning and Processing Language (Second Edition)

Alfred V. Aho

Brian W. Kernighan

Peter J. Weinberger

Bell Laboratories
Murray Hill, New Jersey 07974

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.

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

† UNIX is a trademark of Bell Laboratories.

```
$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]
```

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 new-line, 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 (see 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 the variable 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 new-lines 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 new-line. 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 new-lines 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 %10ld", $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.

Any expression can be used as a subscript in an array reference. Thus

```
x[$1] = $2
```

uses the first field of a record (as a string) to index the array **x**.

Suppose each line of input contains two fields, a name and a non-zero value. Names may be repeated; the task is to print a list of each unique name followed by the sum of all the values for that name. This can be done with the program

```
    { amount[$1] += $2 }
END { for (name in amount)
        print name, amount[name] }
```

To sort the output, replace the last line by

```
print name, amount[name] | "sort"
```

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] T. A. Dolotta, S. B. Olsson, and A. G. Petruccelli (eds). *UNIX User's Manual*—Release 3.0, Bell Laboratories (June 1980).

[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*, Bell Laboratories, 1975.

[4] S. C. Johnson. *YACC—Yet Another Compiler-Compiler*, Bell Laboratories, 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 (in Seconds).

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    ;
```

January 1981

Source Code Control System User's Guide

L. E. Bonanni
C. A. Salemi

Bell Laboratories
Piscataway, New Jersey 08854

ABSTRACT

The Source Code Control System (SCCS) is a system for controlling changes to files of text (typically, the source code and documentation of software systems). It provides facilities for storing, updating, and retrieving any version of a file of text, for controlling updating privileges to that file, for identifying the version of a retrieved file, and for recording who made each change, when and where it was made, and why. SCCS is a collection of programs that run under the UNIX† Time-Sharing System.

This document, together with relevant portions of the *UNIX User's Manual,* is a complete user's guide to SCCS, and supersedes all previous versions. The following topics are covered:

- How to get started with SCCS.
- The scheme used to identify versions of text kept in an SCCS file.
- Basic information needed for day-to-day use of SCCS commands, including a discussion of the more useful arguments.
- Protection and auditing of SCCS files, including the differences between the use of SCCS by *individual* users on one hand, and *groups* of users on the other.

Neither the implementation of SCCS nor the installation procedure for SCCS are described here.

1. INTRODUCTION

The Source Code Control System (SCCS) is a collection of UNIX commands that help individuals or projects control and account for changes to files of text (typically, the source code and documentation of software systems). It is convenient to conceive of SCCS as a custodian of files; it allows retrieval of particular versions of the files, administers changes to them, controls updating privileges to them, and records who made each change, when and where it was made, and why. This is important when programs and documentation undergo frequent changes (because of maintenance and/or enhancement work), inasmuch as it is sometimes desirable to regenerate the version of a program or document as it was before changes were applied to it. Obviously, this could be done by keeping copies (on paper or other media), but this quickly becomes unmanageable and wasteful as the number of programs and documents increases. SCCS provides an attractive solution because it stores on disk the original file and, whenever changes are made to it, stores only the *changes;* each set of changes is called a "delta."

This document, together with relevant portions of the *UNIX User's Manual,* is a complete user's guide to SCCS. This manual contains the following sections:

- *SCCS for Beginners:* How to make an SCCS file, how to update it, and how to retrieve a version thereof.
- *How Deltas Are Numbered:* How versions of SCCS files are numbered and named.

---

† UNIX is a trademark of Bell Laboratories.

- *SCCS Command Conventions:* Conventions and rules generally applicable to all SCCS commands.
- *SCCS Commands:* Explanation of all SCCS commands, with discussions of the more useful arguments.
- *SCCS Files:* Protection, format, and auditing of SCCS files, including a discussion of the differences between using SCCS as an individual and using it as a member of a group or project. The role of a "project SCCS administrator" is introduced.

2. SCCS FOR BEGINNERS

It is assumed that the reader knows how to log onto a UNIX system, create files, and use the text editor. A number of terminal-session fragments are presented below. All of them should be tried: the best way to learn SCCS is to use it.

To supplement the material in this manual, the detailed SCCS command descriptions (appearing in the *UNIX User's Manual*) should be consulted. Section 5 below contains a list of all the SCCS commands. For the time being, however, only basic concepts will be discussed.

2.1 Terminology

Each SCCS file is composed of one or more sets of changes applied to the null (empty) version of the file, with each set of changes usually depending on all previous sets. Each set of changes is called a "delta" and is assigned a name, called the *S*CCS *ID*entification string (SID), composed of at most four components, only the first two of which will concern us for now; these are the "release" and "level" numbers, separated by a period. Hence, the first delta is called "1.1", the second "1.2", the third "1.3", etc. The release number can also be changed allowing, for example, deltas "2.1", "3.19", etc. The change in the release number usually indicates a major change to the file.

Each delta of an SCCS file defines a particular version of the file. For example, delta 1.5 defines version 1.5 of the SCCS file, obtained by applying to the null (empty) version of the file the changes that constitute deltas 1.1, 1.2, etc., up to and including delta 1.5 itself, in that order.

2.2 Creating an SCCS File: the "admin" Command

Consider, for example, a file called "lang" that contains a list of programming languages:

```
c
pl/i
fortran
cobol
algol
```

We wish to give custody of this file to SCCS. The following *admin* command (which is used to *administer* SCCS files) creates an SCCS file and initializes delta 1.1 from the file "lang":

```
admin -ilang s.lang
```

All SCCS files *must* have names that begin with "s.", hence, "s.lang". The **-i** keyletter, together with its value "lang", indicates that *admin* is to create a new SCCS file and *initialize* it with the contents of the file "lang". This initial version is a set of changes applied to the null SCCS file; it is delta 1.1.

The *admin* command replies:

```
No id keywords (cm7)
```

This is a warning message (which may also be issued by other SCCS commands) that is to be ignored for the purposes of this section. Its significance is described in Section 5.1 below. In the following examples, this warning message is not shown, although it may actually be issued by the various command.

The file "lang" should be removed (because it can be easily reconstructed by using the *get* command, below):

```
rm lang
```

2.3 Retrieving a File: the "get" Command

The command:

```
get s.lang
```

causes the creation (retrieval) of the latest version of file "s.lang", and prints the following messages:

```
1.1
5 lines
```

This means that *get* retrieved version 1.1 of the file, which is made up of 5 lines of text. The retrieved text is placed in a file whose name is formed by deleting the "s." prefix from the name of the SCCS file; hence, the file "lang" is created.

The above *get* command simply creates the file "lang" read-only, and keeps no information whatsoever regarding its creation. On the other hand, in order to be able to subsequently apply changes to an SCCS file with the *delta* command (see below), the *get* command must be informed of your intention to do so. This is done as follows:

```
get -e s.lang
```

The -e keyletter causes *get* to create a file "lang" for both reading and writing (so that it may be edited) and places certain information about the SCCS file in another new file, called the *p-file*, that will be read by the *delta* command. The *get* command prints the same messages as before, except that the SID of the version to be created through the use of *delta* is also issued. For example:

```
get -e s.lang
1.1
new delta 1.2
5 lines
```

The file "lang" may now be changed, for example, by:

```
ed lang
27
$a
snobol
ratfor
.
w
41
q
```

2.4 Recording Changes: the "delta" Command

In order to record within the SCCS file the changes that have been applied to "lang", execute:

```
delta s.lang
```

Delta prompts with:

```
comments?
```

the response to which should be a description of why the changes were made; for example:

comments? added more languages

Delta then reads the *p-file*, and determines what changes were made to the file "lang". It does this by doing its own *get* to retrieve the original version, and by applying *diff*(1)[1] to the original version and the edited version.

When this process is complete, at which point the changes to "lang" have been stored in "s.lang", *delta* outputs:

1.2
2 inserted
0 deleted
5 unchanged

The number "1.2" is the name of the delta just created, and the next three lines of output refer to the number of lines in the file "s.lang".

2.5 More about the "get" Command

As we have seen:

get s.lang

retrieves the latest version (now 1.2) of the file "s.lang". This is done by starting with the original version of the file and successively applying deltas (the changes) in order, until all have been applied.

For our example, the following commands are all equivalent:

get s.lang

get −r1 s.lang

get −r1.2 s.lang

The numbers following the **−r** keyletter are SIDs (see Section 2.1 above). Note that omitting the level number of the SID (as in the second example above) is equivalent to specifying the *highest* level number that exists within the specified release. Thus, the second command requests the retrieval of the latest version in release 1, namely 1.2. The third command specifically requests the retrieval of a particular version, in this case, also 1.2.

Whenever a truly major change is made to a file, the significance of that change is usually indicated by changing the *release* number (first component of the SID) of the delta being made. Since normal, automatic, numbering of deltas proceeds by incrementing the level number (second component of the SID), we must indicate to SCCS that we wish to change the release number. This is done with the *get* command:

get −e −r2 s.lang

Because release 2 does not exist, *get* retrieves the latest version *before* release 2; it also interprets this as a request to change the release number of the delta we wish to create to 2, thereby causing it to be named 2.1, rather than 1.3. This information is conveyed to *delta* via the *p-file*. *Get* then outputs:

1. All references of the form *name*(*N*) refer to item *name* in Section *N* of *UNIX User's Manual*.

```
1.2
new delta 2.1
7 lines
```

which indicates that version 1.2 has been retrieved and that 2.1 is the version *delta* will create. If the file is now edited, for example, by:

```
ed lang
41
/cobol/d
w
35
q
```

and *delta* executed:

```
delta s.lang
comments? deleted cobol from list of languages
```

we will see, by *delta's* output, that version 2.1 is indeed created:

```
2.1
0 inserted
1 deleted
6 unchanged
```

Deltas may now be created in release 2 (deltas 2.2, 2.3, etc.), or another new release may be created in a similar manner. This process may be continued as desired.

2.6 The "help" Command

If the command:

```
get abc
```

is executed, the following message will be output:

```
ERROR [abc]: not an SCCS file (co1)
```

The string "co1" is a code for the diagnostic message, and may be used to obtain a fuller explanation of that message by use of the *help* command:

```
help co1
```

This produces the following output:

```
co1:
"not an SCCS file"
A file that you think is an SCCS file
does not begin with the characters "s.".
```

Thus, *help* is a useful command to use whenever there is any doubt about the meaning of an SCCS message. Fuller explanations of almost all SCCS messages may be found in this manner.

3. HOW DELTAS ARE NUMBERED

It is convenient to conceive of the deltas applied to an SCCS file as the nodes of a tree, in which the root is the initial version of the file. The root delta (node) is normally named "1.1" and successor deltas (nodes) are named "1.2", "1.3", etc. The components of the names of the deltas are called the "release" and the "level" numbers, respectively. Thus, normal naming of successor deltas proceeds by incrementing the level number, which is performed automatically by SCCS whenever a delta is made. In addition, the user may wish to change the *release* number when making a delta, to indicate that a major change is being made. When this is

done, the release number also applies to all successor deltas, unless specifically changed again. Thus, the evolution of a particular file may be represented as in Figure 1.

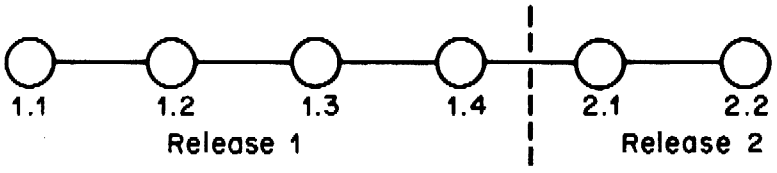


Figure 1. Evolution of an SCCS File

Such a structure may be termed the "trunk" of the SCCS tree. It represents the normal *sequential* development of an SCCS file, in which changes that are part of any given delta are dependent upon *all* the preceding deltas.

However, there are situations in which it is necessary to cause a *branching* in the tree, in that changes applied as part of a given delta are *not* dependent upon all previous deltas. As an example, consider a program which is in production use at version 1.3, and for which development work on release 2 is already in progress. Thus, release 2 may already have some deltas, precisely as shown in Figure 1. Assume that a production user reports a problem in version 1.3, and that the nature of the problem is such that it cannot wait to be repaired in release 2. The changes necessary to repair the trouble will be applied as a delta to version 1.3 (the version in production use). This creates a new version that will then be released to the user, but will *not* affect the changes being applied for release 2 (i.e., deltas 1.4, 2.1, 2.2, etc.).

The new delta is a node on a "branch" of the tree, and its name consists of *four* components, namely, the release and level numbers, as with trunk deltas, plus the "branch" and "sequence" numbers, as follows:

release.level.branch.sequence

The *branch* number is assigned to each branch that is a descendant of a particular trunk delta, with the first such branch being 1, the next one 2, and so on. The *sequence* number is assigned, in order, to each delta on a *particular branch*. Thus, 1.3.1.2 identifies the second delta of the first branch that derives from delta 1.3. This is shown in Figure 2.

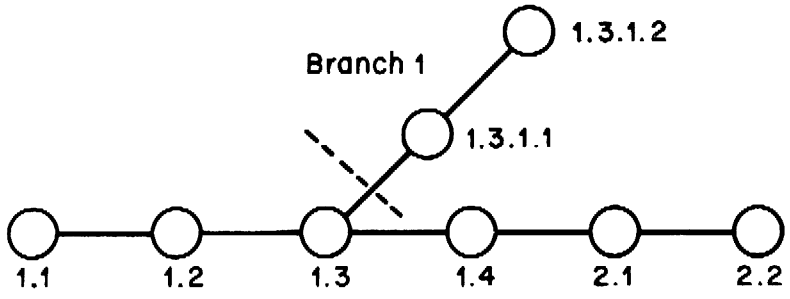


Figure 2. Tree Structure with Branch Deltas

The concept of branching may be extended to any delta in the tree; the naming of the resulting deltas proceeds in the manner just illustrated.

Two observations are of importance with regard to naming deltas. First, the names of trunk deltas contain exactly two components, and the names of branch deltas contain exactly four components. Second, the first two components of the name of branch deltas are always those of the ancestral trunk delta, and the branch component is assigned in the order of creation of the branch, independently of its location relative to the trunk delta. Thus, a branch delta may

always be identified as such from its name. Although the ancestral trunk delta may be identified from the branch delta's name, it is *not* possible to determine the *entire* path leading from the trunk delta to the branch delta. For example, if delta 1.3 has one branch emanating from it, all deltas on that branch will be named 1.3.1.*n*. If a delta on this branch then has another branch emanating from *it*, all deltas on the new branch will be named 1.3.2.*n* (see Figure 3). The only information that may be derived from the name of delta 1.3.2.2 is that it is the *chronologically* second delta on the *chronologically* second branch whose *trunk* ancestor is delta 1.3. In particular, it is *not* possible to determine from the name of delta 1.3.2.2 all of the deltas between it and its trunk ancestor (1.3).

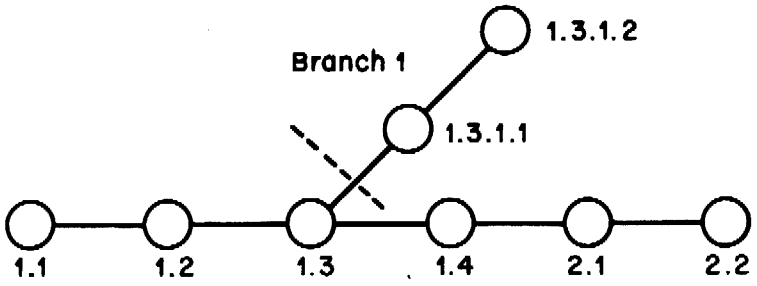


Figure 3. Extending the Branching Concept

It is obvious that the concept of branch deltas allows the generation of arbitrarily complex tree structures. Although this capability has been provided for certain specialized uses, it is strongly recommended that the SCCS tree be kept as simple as possible, because comprehension of its structure becomes extremely difficult as the tree becomes more complex.

4. SCCS COMMAND CONVENTIONS

This section discusses the conventions and rules that apply to SCCS commands. These rules and conventions are generally applicable to *all* SCCS commands, except as indicated below. SCCS commands accept two types of arguments: *keyletter* arguments and *file* arguments.

Keyletter arguments (hereafter called simply "keyletters") begin with a minus sign (−), followed by a lower-case alphabetic character, and, in some cases, followed by a value. These keyletters control the execution of the command to which they are supplied.

File arguments (which may be names of files and/or directories) specify the file(s) that the given SCCS command is to process; naming a directory is equivalent to naming *all* the SCCS files within the directory. Non-SCCS files and unreadable[2] files in the named directories are silently ignored.

In general, file arguments may *not* begin with a minus sign. However, if the name "−" (a lone minus sign) is specified as an argument to a command, the command reads the standard input for lines and takes each line as the *name* of an SCCS file to be processed. The standard input is read until end-of-file. This feature is often used in pipelines with, for example, the *find*(1) or *ls*(1) commands. Again, names of non-SCCS files and of unreadable files are silently ignored.

2. Because of permission modes (see *chmod*(1)).

All keyletters specified for a given command apply to *all* file arguments of that command. All keyletters are processed before any file arguments, with the result that the placement of keyletters is arbitrary (i.e., keyletters may be interspersed with file arguments). File arguments, however, are processed left to right.

Somewhat different argument conventions apply to the *help, what, sccsdiff,* and *val* commands (see Sections 5.5, 5.8, 5.9, and 5.11).

Certain actions of various SCCS commands are controlled by *flags* appearing in SCCS files. Some of these flags are discussed below. For a complete description of all such flags, see *admin*(1).

The distinction between the *real user* (see *passwd*(1)) and the *effective user* of a UNIX system is of concern in discussing various actions of SCCS commands. For the present, it is assumed that both the real user and the effective user are one and the same (i.e., the user who is logged into a UNIX system); this subject is further discussed in Section 6.1.

All SCCS commands that modify an SCCS file do so by writing a temporary copy, called the *x-file,* which ensures that the SCCS file will not be damaged should processing terminate abnormally. The name of the *x-file* is formed by replacing the "s." of the SCCS file name with "x.". When processing is complete, the old SCCS file is removed and the *x-file* is renamed to be the SCCS file. The *x-file* is created in the directory containing the SCCS file, is given the same mode (see *chmod*(1)) as the SCCS file, and is owned by the effective user.

To prevent simultaneous updates to an SCCS file, commands that modify SCCS files create a *lock-file,* called the *z-file,* whose name is formed by replacing the "s." of the SCCS file name with "z.". The *z-file* contains the *process number* of the command that creates it, and its existence is an indication to other commands that that SCCS file is being updated. Thus, other commands that modify SCCS files will not process an SCCS file if the corresponding *z-file* exists. The *z-file* is created with mode 444 (read-only) in the directory containing the SCCS file, and is owned by the effective user. This file exists only for the duration of the execution of the command that creates it. In general, users can ignore *x-files* and *z-files;* they may be useful in the event of system crashes or similar situations.

SCCS commands produce diagnostics (on the diagnostic output) of the form:

ERROR [name-of-file-being-processed]: message text (code)

The *code* in parentheses may be used as an argument to the *help* command (see Section 5.5) to obtain a further explanation of the diagnostic message.

Detection of a fatal error during the processing of a file causes the SCCS command to terminate processing of *that* file and to proceed with the next file, in order, if more than one file has been named.

5. SCCS COMMANDS

This section describes the major features of all the SCCS commands. Detailed descriptions of the commands and of all their arguments are given in the *UNIX User's Manual,* and should be consulted for further information. The discussion below covers only the more common arguments of the various SCCS commands.

Because the commands *get* and *delta* are the most frequently used, they are presented first. The other commands follow in approximate order of importance.

The following is a summary of all the SCCS commands and of their major functions:

get	Retrieves versions of SCCS files.
delta	Applies changes (deltas) to the text of SCCS files, i.e., creates new versions.

admin	Creates SCCS files and applies changes to parameters of SCCS files.
prs	Prints portions of an SCCS file in user specified format.
help	Gives explanations of diagnostic messages.
rmdel	Removes a delta from an SCCS file; allows the removal of deltas that were created by mistake.
cdc	Changes the commentary associated with a delta.
what	Searches any UNIX file(s) for all occurrences of a special pattern and prints out what follows it; is useful in finding identifying information inserted by the *get* command.
sccsdiff	Shows the differences between any two versions of an SCCS file.
comb	Combines two or more consecutive deltas of an SCCS file into a single delta; often reduces the size of the SCCS file.
val	Validates an SCCS file.

5.1 get

The *get* command creates a text file that contains a particular version of an SCCS file. The particular version is retrieved by beginning with the initial version, and then applying deltas, in order, until the desired version is obtained. The created file is called the *g-file;* its name is formed by removing the "s." from the SCCS file name. The *g-file* is created in the current directory and is owned by the real user. The mode assigned to the *g-file* depends on how the *get* command is invoked, as discussed below.

The most common invocation of *get* is:

```
get s.abc
```

which normally retrieves the latest version on the trunk of the SCCS file tree, and produces (for example) on the standard output:

```
1.3
67 lines
No id keywords (cm7)
```

which indicates that:

1. Version 1.3 of file "s.abc" was retrieved (1.3 is the latest trunk delta).
2. This version has 67 lines of text.
3. No ID keywords were substituted in the file (see Section 5.1.1 for a discussion of ID keywords).

The generated *g-file* (file "abc") is given mode 444 (read-only), since this particular way of invoking *get* is intended to produce *g-files* only for inspection, compilation, etc., and *not* for editing (i.e., *not* for making deltas).

In the case of several file arguments (or directory-name arguments), similar information is given for each file processed, but the SCCS file name precedes it. For example:

```
get s.abc s.def
```

produces:

```
s.abc:
1.3
67 lines
No id keywords (cm7)

s.def:
1.7
85 lines
No id keywords (cm7)
```

5.1.1 ID Keywords

In generating a *g-file* to be used for compilation, it is useful and informative to record the date and time of creation, the version retrieved, the module's name, etc., within the *g-file*, so as to have this information appear in a load module when one is eventually created. SCCS provides a convenient mechanism for doing this automatically. *Identification (ID) keywords* appearing anywhere in the generated file are replaced by appropriate values according to the definitions of these ID keywords. The format of an ID keyword is an upper-case letter enclosed by percent signs (%). For example:

```
%I%
```

is defined as the ID keyword that is replaced by the SID of the retrieved version of a file. Similarly, **%H%** is defined as the ID keyword for the current date (in the form "mm/dd/yy"), and **%M%** is defined as the name of the *g-file*. Thus, executing *get* on an SCCS file that contains the PL/I declaration:

```
DCL ID CHAR(100) VAR INIT('%M% %I% %H%');
```

gives (for example) the following:

```
DCL ID CHAR(100) VAR INIT('MODNAME 2.3 07/07/77');
```

When no ID keywords are substituted by *get*, the following message is issued:

```
No id keywords (cm7)
```

This message is normally treated as a warning by *get*, although the presence of the **i** flag in the SCCS file causes it to be treated as an error (see Section 5.2 for further information).

For a complete list of the approximately twenty ID keywords provided, see *get*(1).

5.1.2 Retrieval of Different Versions

Various keyletters are provided to allow the retrieval of other than the default version of an SCCS file. Normally, the default version is the most recent delta of the highest-numbered release on the *trunk* of the SCCS file tree. However, if the SCCS file being processed has a **d** (default SID) flag, the SID specified as the value of this flag is used as a default. The default SID is interpreted in exactly the same way as the value supplied with the **−r** keyletter of *get*.

The **−r** keyletter is used to specify an SID to be retrieved, in which case the **d** (default SID) flag (if any) is ignored. For example:

```
get −r1.3 s.abc
```

retrieves version 1.3 of file "s.abc", and produces (for example) on the standard output:

```
1.3
64 lines
```

A branch delta may be retrieved similarly:

```
get −r1.5.2.3 s.abc
```

which produces (for example) on the standard output:

```
1.5.2.3
234 lines
```

When a two- or four-component SID is specified as a value for the **−r** keyletter (as above) and the particular version does not exist in the SCCS file, an error message results. Omission of the level number, as in:

```
get −r3 s.abc
```

causes retrieval of the *trunk* delta with the highest level number within the given release, if the given release exists. Thus, the above command might output:

```
3.7
213 lines
```

If the given release does not exist, *get* retrieves the *trunk* delta with the highest level number within the highest-numbered existing release that is lower than the given release. For example, assuming release 9 does not exist in file "s.abc", and that release 7 is actually the highest-numbered release below 9, execution of:

```
get −r9 s.abc
```

might produce:

```
7.6
420 lines
```

which indicates that trunk delta 7.6 is the latest version of file "s.abc" below release 9. Similarly, omission of the sequence number, as in:

```
get −r4.3.2 s.abc
```

results in the retrieval of the branch delta with the highest sequence number on the given branch, if it exists. (If the given branch does not exist, an error message results.) This might result in the following output:

```
4.3.2.8
89 lines
```

The **−t** keyletter is used to retrieve the latest ("top") version in a particular *release* (i.e., when no **−r** keyletter is supplied, or when its value is simply a release number). The latest version is defined as that delta which was produced most recently, independent of its location on the SCCS file tree. Thus, if the most recent delta in release 3 is 3.5,

```
get −r3 −t s.abc
```

might produce:

```
3.5
59 lines
```

However, if branch delta 3.2.1.5 were the latest delta (created after delta 3.5), the same command might produce:

```
3.2.1.5
46 lines
```

5.1.3 Retrieval with Intent to Make a Delta

Specification of the **−e** keyletter to the *get* command is an indication of the intent to make a delta, and, as such, its use is restricted. The presence of this keyletter causes *get* to check:

1. The *user list* (which is the list of *login* names and/or *group IDs* of users allowed to make deltas (see Section 6.2)) to determine if the login name or group ID of the user executing *get* is on that list. Note that a *null* (empty) user list behaves as if it contained *all* possible login names.
2. That the *release* (R) of the version being retrieved satisfies the relation:

   floor $\leq$ R $\leq$ ceiling

   to determine if the release being accessed is a protected release. The *floor* and *ceiling* are specified as *flags* in the SCCS file.
3. That the *release* (R) is not *locked* against editing. The *lock* is specified as a flag in the SCCS file.
4. Whether or not *multiple concurrent edits* are allowed for the SCCS file as specified by the **j** flag in the SCCS file (multiple concurrent edits are described in Section 5.1.5).

A failure of any of the first three conditions causes the processing of the corresponding SCCS file to terminate.

If the above checks succeed, the **−e** keyletter causes the creation of a *g-file* in the current directory with mode 644 (readable by everyone, writable only by the owner) owned by the real user. If a *writable g-file* already exists, *get* terminates with an error. This is to prevent inadvertent destruction of a *g-file* that already exists and is being edited for the purpose of making a delta.

Any ID keywords appearing in the *g-file* are *not* substituted by *get* when the **−e** keyletter is specified, because the generated *g-file* is to be subsequently used to create another delta, and replacement of ID keywords would cause them to be permanently changed within the SCCS file. In view of this, *get* does not need to check for the presence of ID keywords within the *g-file*, so that the message:

```
No id keywords (cm7)
```

is never output when *get* is invoked with the **−e** keyletter.

In addition, the **−e** keyletter causes the creation (or updating) of a *p-file*, which is used to pass information to the *delta* command (see Section 5.1.4).

The following is an example of the use of the **−e** keyletter:

```
get −e s.abc
```

which produces (for example) on the standard output:

```
1.3
new delta 1.4
67 lines
```

If the **−r** and/or **−t** keyletters are used together with the **−e** keyletter, the version retrieved for editing is as specified by the **−r** and/or **−t** keyletters.

The keyletters **−i** and **−x** may be used to specify a list (see *get*(1) for the syntax of such a list) of deltas to be *included* and *excluded*, respectively, by *get*. Including a delta means forcing the changes that constitute the particular delta to be included in the retrieved version. This is useful if one wants to apply the same changes to more than one version of the SCCS file. Excluding a delta means forcing it to be *not* applied. This may be used to undo, in the version of the SCCS file to be created, the effects of a previous delta. Whenever deltas are included or excluded, *get* checks for possible interference between such deltas and those deltas that are normally used in retrieving the particular version of the SCCS file. (Two deltas can interfere, for example, when each one changes the same line of the retrieved *g-file*.) Any interference is indicated by a warning that shows the range of lines within the retrieved *g-file* in which the problem may exist. The user is expected to examine the *g-file* to determine whether a problem

actually exists, and to take whatever corrective measures (if any) are deemed necessary (e.g., edit the file).

☞ *The* **−i** *and* **−x** *keyletters should be used with extreme care.*

The **−k** keyletter is provided to facilitate regeneration of a *g-file* that may have been accidentally removed or ruined subsequent to the execution of *get* with the **−e** keyletter, or to simply generate a *g-file* in which the replacement of ID keywords has been suppressed. Thus, a *g-file* generated by the **−k** keyletter is identical to one produced by *get* executed with the **−e** keyletter. However, no processing related to the *p-file* takes place.

5.1.4 Concurrent Edits of Different SIDs

The ability to retrieve different versions of an SCCS file allows a number of deltas to be "in progress" at any given time. This means that a number of *get* commands with the **−e** keyletter may be executed on the same file, provided that no two executions retrieve the same version (unless multiple concurrent edits are allowed, see Section 5.1.5).

The *p-file* (which is created by the *get* command invoked with the **−e** keyletter) is named by replacing the "s." in the SCCS file name with "p.". It is created in the directory containing the SCCS file, is given mode 644 (readable by everyone, writable only by the owner), and is owned by the effective user. The *p-file* contains the following information for each delta that is still "in progress":[3]

- The SID of the retrieved version.
- The SID that will be given to the new delta when it is created.
- The login name of the real user executing *get*.

The first execution of "get −e" causes the *creation* of the *p-file* for the corresponding SCCS file. Subsequent executions only *update* the *p-file* with a line containing the above information. Before updating, however, *get* checks that no entry already in the *p-file* specifies as already retrieved the SID of the version to be retrieved, unless multiple concurrent edits are allowed.

If both checks succeed, the user is informed that other deltas are in progress, and processing continues. If either check fails, an error message results. It is important to note that the various executions of *get* should be carried out from different directories. Otherwise, only the first execution will succeed, since subsequent executions would attempt to over-write a *writable g-file,* which is an SCCS error condition. In practice, such multiple executions are performed by different users,[4] so that this problem does not arise, since each user normally has a different working directory.

Table 1 shows, for the most useful cases, what version of an SCCS file is retrieved by *get,* as well as the SID of the version to be eventually created by *delta,* as a function of the SID specified to *get.*

5.1.5 Concurrent Edits of the Same SID

Under normal conditions, *gets* for editing (**−e** keyletter is specified) based on the same SID are not permitted to occur concurrently. That is, *delta* must be executed before a subsequent *get* for editing is executed at the same SID as the previous *get.* However, multiple concurrent edits (defined to be two or more *successive* executions of *get* for editing based on the same retrieved SID) *are* allowed if the **j** flag is set in the SCCS file. Thus:

---

3. Other information may be present, but is not of concern here. See *get*(1) for further discussion.
4. See Section 6.1 for a discussion of how different users are permitted to use SCCS commands on the same files.

TABLE 1. Determination of New SID

Case	*SID Specified**	*−b Keyletter Used†*	*Other Conditions*	*SID Retrieved*	*SID of Delta to be Created*
1.	none‡	no	R defaults to mR	mR.mL	mR.(mL +1)
2.	none‡	yes	R defaults to mR	mR.mL	mR.mL.(mB +1).1
3.	R	no	R > mR	mR.mL	R.1§
4.	R	no	R = mR	mR.mL	mR.(mL +1)
5.	R	yes	R > mR	mR.mL	mR.mL.(mB +1).1
6.	R	yes	R = mR	mR.mL	mR.mL.(mB +1).1
7.	R	−	R < mR and R does *not* exist	hR.mL**	hR.mL.(mB +1).1
8.	R	−	Trunk successor in release > R and R exists	R.mL	R.mL.(mB +1).1
9.	R.L	no	No trunk successor	R.L	R.(L +1)
10.	R.L	yes	No trunk successor	R.L	R.L.(mB +1).1
11.	R.L	−	Trunk successor in release ≥ R	R.L	R.L.(mB +1).1
12.	R.L.B	no	No branch successor	R.L.B.mS	R.L.B.(mS +1)
13.	R.L.B	yes	No branch successor	R.L.B.mS	R.L.(mB +1).1
14.	R.L.B.S	no	No branch successor	R.L.B.S	R.L.B.(S +1)
15.	R.L.B.S	yes	No branch successor	R.L.B.S	R.L.(mB +1).1
16.	R.L.B.S	−	Branch successor	R.L.B.S	R.L.(mB +1).1

* "R", "L", "B", and "S" are the "release", "level", "branch", and "sequence" components of the SID, respectively; "m" means "maximum". Thus, for example, "R.mL" means "the maximum level number within release R"; "R.L.(mB +1).1" means "the first sequence number on the *new* branch (i.e., maximum branch number plus 1) of level L within release R". Note that if the SID specified is of the form "R.L", "R.L.B", or "R.L.B.S", each of the specified components *must* exist.

† The **−b** keyletter is effective only if the **b** flag (see *admin*(1)) is present in the file. In this table, an entry of "−" means "irrelevant".

‡ This case applies if the **d** (default SID) flag is *not* present in the file. If the **d** flag *is* present in the file, then the SID obtained from the **d** flag is interpreted as if it had been specified on the command line. Thus, one of the other cases in this table applies.

§ This case is used to force the creation of the *first* delta in a *new* release.

** "hR" is the highest *existing* release that is lower than the specified, *nonexistent*, release R.

```
get −e s.abc
1.1
new delta 1.2
5 lines
```

may be immediately followed by:

```
get −e s.abc
1.1
new delta 1.1.1.1
5 lines
```

without an intervening execution of *delta*. In this case, a *delta* command corresponding to the first *get* produces delta 1.2 (assuming 1.1 is the latest (most recent) trunk delta), and the *delta* command corresponding to the second *get* produces delta 1.1.1.1.

5.1.6 Keyletters that Affect Output

Specification of the **−p** keyletter causes *get* to write the retrieved text to the standard output, rather than to a *g-file*. In addition, all output normally directed to the standard output (such as the SID of the version retrieved and the number of lines retrieved) is directed instead to the diagnostic output. This may be used, for example, to create *g-files* with arbitrary names:

```
get −p s.abc > arbitrary-file-name
```

The **−p** keyletter is particularly useful when used with the "!" or "$" arguments of the UNIX *send* (1C) command. For example:

```
send MOD=s.abc REL=3 compile
```

given that file "compile" contains:

```
//plicomp job job-card-information
//step1 exec plickc
//pli.sysin dd *
~−s
~!get −p −rREL MOD
/*
//
```

will *send* the highest level of release 3 of file "s.abc". Note that the line "~−s", which causes *send*~(1C) to make ID keyword substitutions before detecting and interpreting control lines, is necessary if *send*~(1C) is to substitute "s.abc" for MOD and "3" for REL in the line "~!get −p −rREL MOD".

The **−s** keyletter suppresses all output that is *normally* directed to the standard output. Thus, the SID of the retrieved version, the number of lines retrieved, etc., are not output. This does not, however, affect messages to the diagnostic output. This keyletter is used to prevent non-diagnostic messages from appearing on the user's terminal, and is often used in conjunction with the **−p** keyletter to "pipe" the output of *get*, as in:

```
get −p −s s.abc | nroff
```

The **−g** keyletter is supplied to suppress the actual retrieval of the text of a version of the SCCS file. This may be useful in a number of ways. For example, to verify the existence of a particular SID in an SCCS file, one may execute:

```
get −g −r4.3 s.abc
```

This outputs the given SID if it exists in the SCCS file, or it generates an error message, if it does not. Another use of the **−g** keyletter is in regenerating a *p-file* that may have been accidentally destroyed:

```
get −e −g s.abc
```

The **−l** keyletter causes the creation of an *l-file*, which is named by replacing the "s." of the SCCS file name with "l.". This file is created in the current directory, with mode 444 (read-only), and is owned by the real user. It contains a table (whose format is described in *get*(1)) showing which deltas were used in constructing a particular version of the SCCS file. For example:

```
get −r2.3 −l s.abc
```

generates an *l-file* showing which deltas were applied to retrieve version 2.3 of the SCCS file. Specifying a *value* of "p" with the **−l** keyletter, as in:

```
get −lp −r2.3 s.abc
```

causes the generated output to be written to the standard output rather than to the *l-file*. The **−g** keyletter may be used with the **−l** keyletter to suppress the actual retrieval of the text.

The **−m** keyletter is of use in identifying, line by line, the changes applied to an SCCS file. Specification of this keyletter causes each line of the generated *g-file* to be preceded by the SID of the delta that caused that line to be inserted. The SID is separated from the text of the line by a tab character.

The **−n** keyletter causes each line of the generated *g-file* to be preceded by the value of the %M% ID keyword (see Section 5.1.1) and a tab character. The **−n** keyletter is most often used in a pipeline with *grep*(1). For example, to find all lines that match a given pattern in the latest version of each SCCS file in a directory, the following may be executed:

```
get −p −n −s directory | grep pattern
```

If both the **−m** and **−n** keyletters are specified, each line of the generated *g-file* is preceded by the value of the %M% ID keyword and a tab (this is the effect of the **−n** keyletter), followed by the line in the format produced by the **−m** keyletter. Because use of the **−m** keyletter and/or the **−n** keyletter causes the contents of the *g-file* to be modified, such a *g-file* must *not* be used for creating a delta. Therefore, neither the **−m** keyletter nor the **−n** keyletter may be specified together with the **−e** keyletter.

See *get*(1) for a full description of additional *get* keyletters.

5.2 delta

The *delta* command is used to incorporate the changes made to a *g-file* into the corresponding SCCS file, i.e., to create a delta, and, therefore, a new version of the file.

Invocation of the *delta* command requires the existence of a *p-file* (see Sections 5.1.3 and 5.1.4). *Delta* examines the *p-file* to verify the presence of an entry containing the user's login name. If none is found, an error message results. *Delta* also performs the same permission checks that *get* performs when invoked with the **−e** keyletter. If all checks are successful, *delta* determines what has been changed in the *g-file*, by comparing it (via *diff*(1)) with its own, temporary copy of the *g-file* as it was before editing. This temporary copy of the *g-file* is called the *d-file* (its name is formed by replacing the "s." of the SCCS file name with "d.") and is obtained by performing an internal *get* at the SID specified in the *p-file* entry.

The required *p-file* entry is the one containing the login name of the user executing *delta*, because the user who retrieved the *g-file* must be the one who will create the delta. However, if the login name of the user appears in more than one entry (i.e., the same user executed *get* with the **−e** keyletter more than once on the same SCCS file), the **−r** keyletter must be used with *delta* to specify an SID that uniquely identifies the *p-file* entry[5]. This entry is the one used to obtain the SID of the delta to be created.

In practice, the most common invocation of *delta* is:

```
delta s.abc
```

which prompts on the standard output (but only if it is a terminal):

```
comments?
```

to which the user replies with a description of why the delta is being made, terminating the reply with a new-line character. The user's response may be up to 512 characters long, with new-lines *not* intended to terminate the response escaped by "\".

---

5. The SID specified may be either the SID retrieved by *get*, or the SID *delta* is to create.

If the SCCS file has a **v** flag, *delta* first prompts with:

MRs?

on the standard output. (Again, this prompt is printed only if the standard output is a terminal.) The standard input is then read for MR[6] numbers, separated by blanks and/or tabs, terminated in the same manner as the response to the prompt "comments?".

The **−y** and/or **−m** keyletters may be used to supply the commentary (comments and MR numbers, respectively) on the command line, rather than through the standard input:

delta −y"descriptive comment" −m"mrnum1 mrnum2" s.abc

In this case, the corresponding prompts are not printed, and the standard input is not read. The **−m** keyletter is allowed only if the SCCS file has a **v** flag. These keyletters are useful when *delta* is executed from within a *shell procedure* (see *sh* (1)).

The commentary (comments and/or MR numbers), whether solicited by *delta* or supplied via keyletters, is recorded as part of the entry for the delta being created, and applies to *all* SCCS files processed by the same invocation of *delta*. This implies that if *delta* is invoked with more than one file argument, and the first file named has a **v** flag, all files named must have this flag. Similarly, if the first file named does not have this flag, then none of the files named may have it. Any file that does not conform to these rules is not processed.

When processing is complete, *delta* outputs (on the standard output) the SID of the created delta (obtained from the *p-file* entry) and the counts of lines inserted, deleted, and left unchanged by the delta. Thus, a typical output might be:

1.4
14 inserted
7 deleted
345 unchanged

It is possible that the counts of lines reported as inserted, deleted, or unchanged by *delta* do not agree with the user's perception of the changes applied to the *g-file*. The reason for this is that there usually are a number of ways to describe a set of such changes, especially if lines are moved around in the *g-file*, and *delta* is likely to find a description that differs from the user's perception. However, the *total* number of lines of the new delta (the number inserted plus the number left unchanged) should agree with the number of lines in the edited *g-file*.

If, in the process of making a delta, *delta* finds no ID keywords in the edited *g-file*, the message:

No id keywords (cm7)

is issued after the prompts for commentary, but before any other output. This indicates that any ID keywords that may have existed in the SCCS file have been replaced by their values, or deleted during the editing process. This could be caused by creating a delta from a *g-file* that was created by a *get* without the **−e** keyletter (recall that ID keywords are replaced by *get* in that case), or by accidentally deleting or changing the ID keywords during the editing of the *g-file*. Another possibility is that the file may never have had any ID keywords. In any case, it is left up to the user to determine what remedial action is necessary, but the delta is made, unless there is an **i** flag in the SCCS file, indicating that this should be treated as a fatal error. In this last case, the delta is not created.

---

6. In a tightly controlled environment, it is expected that deltas are created only as a result of some trouble report, change request, trouble ticket, etc. (collectively called here Modification Requests, or MRs) and that it is desirable or necessary to record such MR number(s) within each delta.

After processing of an SCCS file is complete, the corresponding *p-file* entry is removed from the *p-file*.[7] If there is only *one* entry in the *p-file*, then the *p-file* itself is removed.

In addition, *delta* removes the edited *g-file*, unless the **−n** keyletter is specified. Thus:

```
delta −n s.abc
```

will keep the *g-file* upon completion of processing.

The **−s** ("silent") keyletter suppresses all output that is normally directed to the standard output, other than the prompts "comments?" and "MRs?". Thus, use of the **−s** keyletter together with the **−y** keyletter (and possibly, the **−m** keyletter) causes *delta* neither to read the standard input nor to write the standard output.

The differences between the *g-file* and the *d-file* (see above), which constitute the delta, may be printed on the standard output by using the **−p** keyletter. The format of this output is similar to that produced by *diff*(1).

5.3 admin

The *admin* command is used to *administer* SCCS files, that is, to create new SCCS files and to change parameters of existing ones. When an SCCS file is created, its parameters are initialized by use of keyletters or are assigned default values if no keyletters are supplied. The same keyletters are used to change the parameters of existing files.

Two keyletters are supplied for use in conjunction with detecting and correcting "corrupted" SCCS files, and are discussed in Section 6.3 below.

Newly-created SCCS files are given mode 444 (read-only) and are owned by the effective user.

Only a user with write permission in the directory containing the SCCS file may use the *admin* command upon that file.

5.3.1 Creation of SCCS Files

An SCCS file may be created by executing the command:

```
admin −ifirst s.abc
```

in which the value ("first") of the **−i** keyletter specifies the name of a file from which the text of the *initial* delta of the SCCS file "s.abc" is to be taken. Omission of the value of the **−i** keyletter indicates that *admin* is to read the standard input for the text of the initial delta. Thus, the command:

```
admin −i s.abc < first
```

is equivalent to the previous example. If the text of the initial delta does not contain ID keywords, the message:

```
No id keywords (cm7)
```

is issued by *admin* as a warning. However, if the same invocation of the command also sets the **i** flag (not to be confused with the **−i** keyletter), the message is treated as an error and the SCCS file is not created. Only *one* SCCS file may be created at a time using the **−i** keyletter.

---

7. All updates to the *p-file* are made to a temporary copy, the *q-file*, whose use is similar to the use of the *x-file*, which is described in Section 4 above.

When an SCCS file is created, the *release* number assigned to its first delta is normally "1", and its *level* number is always "1". Thus, the first delta of an SCCS file is normally "1.1". The **−r** keyletter is used to specify the release number to be assigned to the first delta. Thus:

```
admin −ifirst −r3 s.abc
```

indicates that the first delta should be named "3.1" rather than "1.1". Because this keyletter is only meaningful in creating the first delta, its use is only permitted with the **−i** keyletter.

5.3.2 Inserting Commentary for the Initial Delta

When an SCCS file is created, the user may choose to supply commentary stating the reason for creation of the file. This is done by supplying comments (**−y** keyletter) and/or MR numbers[8] (**−m** keyletter) in exactly the same manner as for *delta*. If comments (**−y** keyletter) are omitted, a comment line of the form:

```
date and time created YY/MM/DD HH:MM:SS by logname
```

is automatically generated.

If it is desired to supply MR numbers (**−m** keyletter), the **v** flag must also be set (using the **−f** keyletter described below). The **v** flag simply determines whether or not MR numbers must be supplied when using any SCCS command that modifies a *delta commentary* (see *sccsfile* (5)) in the SCCS file. Thus:

```
admin −ifirst −mmrnum1 −fv s.abc
```

Note that the **−y** and **−m** keyletters are only effective if a new SCCS file is being created.

5.3.3 Initialization and Modification of SCCS File Parameters

The portion of the SCCS file reserved for *descriptive text* (see Section 6.2) may be initialized or changed through the use of the **−t** keyletter. The descriptive text is intended as a summary of the contents and purpose of the SCCS file, although its contents may be arbitrary, and it may be arbitrarily long.

When an SCCS file is being created and the **−t** keyletter is supplied, it must be followed by the name of a file from which the descriptive text is to be taken. For example, the command:

```
admin −ifirst −tdesc s.abc
```

specifies that the descriptive text is to be taken from file "desc".

When processing an *existing* SCCS file, the **−t** keyletter specifies that the descriptive text (if any) currently in the file is to be *replaced* with the text in the named file. Thus:

```
admin −tdesc s.abc
```

specifies that the descriptive text of the SCCS file is to be replaced by the contents of "desc"; omission of the file name after the **−t** keyletter as in:

```
admin −t s.abc
```

causes the *removal* of the descriptive text from the SCCS file.

The *flags* (see Section 6.2) of an SCCS file may be initialized and changed, or deleted through the use of the **−f** and **−d** keyletters, respectively. The flags of an SCCS file are used to direct certain actions of the various commands. See *admin* (1) for a description of all the flags. For example, the **i** flag specifies that the warning message stating there are no ID keywords

---

8. The creation of an SCCS file may sometimes be the direct result of an MR.

contained in the SCCS file should be treated as an error, and the **d** (default SID) flag specifies the default version of the SCCS file to be retrieved by the *get* command. The **−f** keyletter is used to set a flag and, possibly, to set its value. For example:

```
admin −ifirst −fi −fmmodname s.abc
```

sets the **i** flag and the **m** (module name) flag. The value "modname" specified for the **m** flag is the value that the *get* command will use to replace the %M% ID keyword. (In the absence of the **m** flag, the name of the *g-file* is used as the replacement for the %M% ID keyword.) Note that several **−f** keyletters may be supplied on a single invocation of *admin*, and that **−f** keyletters may be supplied whether the command is creating a new SCCS file or processing an existing one.

The **−d** keyletter is used to delete a flag from an SCCS file, and may only be specified when processing an existing file. As an example, the command:

```
admin −dm s.abc
```

removes the **m** flag from the SCCS file. Several **−d** keyletters may be supplied on a single invocation of *admin*, and may be intermixed with **−f** keyletters.

SCCS files contain a list (*user list*) of login names and/or group IDs of users who are allowed to create deltas (see Sections 5.1.3 and 6.2). This list is empty by default, which implies that *anyone* may create deltas. To add login names and/or group IDs to the list, the **−a** keyletter is used. For example:

```
admin −axyz −awql −a1234 s.abc
```

adds the login names "xyz" and "wql" and the group ID "1234" to the list. The **−a** keyletter may be used whether *admin* is creating a new SCCS file or processing an existing one, and may appear several times. The **−e** keyletter is used in an analogous manner if one wishes to remove ("erase") login names or group IDs from the list.

5.4 prs

Prs is used to print on the standard output all or parts of an SCCS file (see Section 6.2) in a format, called the output *data specification*, supplied by the user via the **−d** keyletter. The data specification is a string consisting of SCCS file *data keywords*[9] interspersed with optional user text.

Data keywords are replaced by appropriate values according to their definitions. For example:

```
:I:
```

is defined as the data keyword that is replaced by the SID of a specified delta. Similarly, **:F:** is defined as the data keyword for the SCCS file name currently being processed, and **:C:** is defined as the comment line associated with a specified delta. All parts of an SCCS file have an associated data keyword. For a complete list of the data keywords, see *prs*(1).

There is no limit to the number of times a data keyword may appear in a data specification. Thus, for example:

```
prs −d":I: this is the top delta for :F: :I:" s.abc
```

may produce on the standard output:

---

9. Not to be confused with *get ID keywords*.

2.1 this is the top delta for s.abc 2.1

Information may be obtained from a single delta by specifying the SID of that delta using the **−r** keyletter. For example:

prs −d":F:: :I: comment line is: :C:" −r1.4 s.abc

may produce the following output:

s.abc: 1.4 comment line is: THIS IS A COMMENT

If the **−r** keyletter is *not* specified, the value of the SID defaults to the most recently created delta.

In addition, information from a *range* of deltas may be obtained by specifying the **−l** or **−e** keyletters. The **−e** keyletter substitutes data keywords for the SID designated via the **−r** keyletter and all deltas created *earlier*. The **−l** keyletter substitutes data keywords for the SID designated via the **−r** keyletter and all deltas created *later*. Thus, the command:

prs −d:I: −r1.4 −e s.abc

may output:

1.4
1.3
1.2.1.1
1.2
1.1

and the command:

prs −d:I: −r1.4 −l s.abc

may produce:

3.3
3.2
3.1
2.2.1.1
2.2
2.1
1.4

Substitution of data keywords for *all* deltas of the SCCS file may be obtained by specifying both the **−e** and **−l** keyletters.

5.5 help

The *help* command prints explanations of SCCS commands and of messages that these commands may print. Arguments to *help*, zero or more of which may be supplied, are simply the names of SCCS commands or the code numbers that appear in parentheses after SCCS messages. If no argument is given, *help* prompts for one. *Help* has no concept of *keyletter* arguments or *file* arguments. Explanatory information related to an argument, if it exists, is printed on the standard output. If no information is found, an error message is printed. Note that each argument is processed independently, and an error resulting from one argument will *not* terminate the processing of the other arguments.

Explanatory information related to a command is a synopsis of the command. For example:

help ge5 rmdel

produces:

ge5:
"nonexistent sid"
The specified sid does not exist in the
given file.
Check for typos.

rmdel:
rmdel −rSID name ...

5.6 rmdel

The *rmdel* command is provided to allow *removal* of a delta from an SCCS file, though its use should be reserved for those cases in which incorrect, global changes were made a part of the delta to be removed.

The delta to be removed must be a "leaf" delta. That is, it must be the latest (most recently created) delta on its branch or on the trunk of the SCCS file tree. In Figure 3, only deltas 1.3.1.2, 1.3.2.2, and 2.2 can be removed; once they are removed, then deltas 1.3.2.1 and 2.1 can be removed, and so on.

To be allowed to remove a delta, the effective user must have write permission in the directory containing the SCCS file. In addition, the real user must either be the one who created the delta being removed, or be the owner of the SCCS file and its directory.

The **−r** keyletter, which is mandatory, is used to specify the *complete* SID of the delta to be removed (i.e., it must have two components for a trunk delta, and four components for a branch delta). Thus:

```
rmdel −r2.3 s.abc
```

specifies the removal of (trunk) delta "2.3" of the SCCS file. Before removal of the delta, *rmdel* checks that the *release* number (R) of the given SID satisfies the relation:

floor ≤ R ≤ ceiling

Rmdel also checks that the SID specified is *not* that of a version for which a *get* for editing has been executed and whose associated *delta* has not yet been made. In addition, the login name or group ID of the user must appear in the file's *user list*, or the *user list* must be empty. Also, the release specified can not be *locked* against editing (i.e., if the **l** flag is set (see *admin*(1)), the release specified *must* not be contained in the list). If these conditions are not satisfied, processing is terminated, and the delta is not removed. After the specified delta has been removed, its type indicator in the *delta table* of the SCCS file (see Section 6.2) is changed from "D" (for "delta") to "R" (for "removed").

5.7 cdc

The *cdc* command is used to *change* a delta's commentary that was supplied when that delta was created. Its invocation is analogous to that of the *rmdel* command, except that the delta to be processed is *not* required to be a leaf delta. For example:

```
cdc −r3.4 s.abc
```

specifies that the commentary of delta "3.4" of the SCCS file is to be changed.

The *new* commentary is solicited by *cdc* in the same manner as that of *delta*. The old commentary associated with the specified delta is kept, but it is preceded by a comment line indicating that it has been changed (i.e., superseded), and the new commentary is entered ahead of this comment line. The "inserted" comment line records the login name of the user executing *cdc* and the time of its execution.

Cdc also allows for the deletion of selected MR numbers associated with the specified delta. This is specified by preceding the selected MR numbers by the character "!". Thus:

```
cdc -r1.4 s.abc
MRs? mrnum3 !mrnum1
comments? deleted wrong MR number and inserted correct MR number
```

inserts "mrnum3" and deletes "mrnum1" for delta 1.4.

5.8 what

The *what* command is used to find identifying information within *any* UNIX file whose name is given as an argument to *what*. Directory names and a name of "-" (a lone minus sign) are *not* treated specially, as they are by other SCCS commands, and no *keyletters* are accepted by the command.

What searches the given file(s) for all occurrences of the string "@(#)", which is the replacement for the %Z% ID keyword (see *get*(1)), and prints (on the standard output) what follows that string until the first double quote ("), greater than (>), backslash (\), new-line, or (non-printing) NUL character. Thus, for example, if the SCCS file "s.prog.c" (which is a C program), contains the following line (the %M% and %I% ID keywords were defined in Section 5.1.1):

```
char id[] "%Z%%M%:%I%";
```

and then the command:

```
get -r3.4 s.prog.c
```

is executed, and finally the resulting *g-file* is compiled to produce "prog.o" and "a.out", then the command:

```
what prog.c prog.o a.out
```

produces:

```
prog.c:
   prog.c:3.4
prog.o:
   prog.c:3.4
a.out:
   prog.c:3.4
```

The string searched for by *what* need not be inserted via an ID keyword of *get;* it may be inserted in any convenient manner.

5.9 sccsdiff

The *sccsdiff* command determines (and prints on the standard output) the differences between two specified versions of one or more SCCS files. The versions to be compared are specified by using the **-r** keyletter, whose format is the same as for the *get* command. The two versions *must* be specified as the first two arguments to this command in the order in which they were created, i.e., the older version is specified first. Any following keyletters are interpreted as arguments to the *pr*(1) command (which actually prints the differences) and must appear before any file names. SCCS files to be processed are named last. Directory names and a name of "-" (a lone minus sign) are *not* acceptable to *sccsdiff*.

The differences are printed in the form generated by *diff*(1). The following is an example of the invocation of *sccsdiff:*

```
sccsdiff -r3.4 -r5.6 s.abc
```

5.10 comb

Comb generates a *shell procedure* (see *sh*(1)) which attempts to reconstruct the named SCCS files so that the reconstructed files are smaller than the originals. The generated shell procedure is written on the standard output.

Named SCCS files are reconstructed by discarding unwanted deltas and combining specified other deltas. The intended use is for those SCCS files that contain deltas that are so old that they are no longer useful. It is *not* recommended that *comb* be used as a matter of routine; its use should be restricted to a *very* small number of times in the life of an SCCS file.

In the absence of any keyletters, *comb* preserves only leaf deltas and the minimum number of ancestor deltas necessary to preserve the "shape" of the SCCS file tree. The effect of this is to eliminate "middle" deltas on the trunk and on all branches of the tree. Thus, in Figure 3, deltas 1.2, 1.3.2.1, 1.4, and 2.1 would be eliminated. Some of the keyletters are summarized as follows:

The **−p** keyletter specifies the oldest delta that is to be preserved in the reconstruction. All older deltas are discarded.

The **−c** keyletter specifies a *list* (see *get*(1) for the syntax of such a list) of deltas to be preserved. All other deltas are discarded.

The **−s** keyletter causes the generation of a shell procedure, which, when run, produces *only* a report summarizing the percentage space (if any) to be saved by reconstructing each named SCCS file. It is recommended that *comb* be run with this keyletter (in addition to any others desired) *before* any actual reconstructions.

It should be noted that the shell procedure generated by *comb* is *not* guaranteed to save any space. In fact, it is possible for the reconstructed file to be *larger* than the original. Note, too, that the shape of the SCCS file tree may be altered by the reconstruction process.

5.11 val

Val is used to determine if a file is an SCCS file meeting the characteristics specified by an optional list of keyletter arguments. Any characteristics not met are considered errors.

Val checks for the existence of a particular delta when the SID for that delta is *explicitly* specified via the **−r** keyletter. The string following the **−y** or **−m** keyletter is used to check the value set by the **t** or **m** flag respectively (see *admin*(1) for a description of the flags).

Val treats the special argument "−" differently from other SCCS commands (see Section 4). This argument allows *val* to read the argument list from the standard input as opposed to obtaining it from the command line. The standard input is read until end-of-file. This capability allows for one invocation of *val* with different values for the keyletter and file arguments. For example:

```
val −
−yc −mabc s.abc
−mxyz −ypl1 s.xyz
```

first checks if file "s.abc" has a value "c" for its *type* flag and value "abc" for the *module name* flag. Once processing of the first file is completed, *val* then processes the remaining files, in this case "s.xyz", to determine if they meet the characteristics specified by the keyletter arguments associated with them.

Val returns an 8-bit code; each bit set indicates the occurrence of a specific error (see *val*(1) for a description of the possible errors and their codes). In addition, an appropriate diagnostic is printed unless suppressed by the **−s** keyletter. A return code of "0" indicates all named files met the characteristics specified.

6. SCCS FILES

This section discusses several topics that must be considered before extensive use is made of SCCS. These topics deal with the protection mechanisms relied upon by SCCS, the format of SCCS files, and the recommended procedures for auditing SCCS files.

6.1 Protection

SCCS relies on the capabilities of the UNIX operating system for most of the protection mechanisms required to prevent unauthorized changes to SCCS files (i.e., changes made by non-SCCS commands). The only protection features provided directly by SCCS are the *release lock* flag, the *release floor* and *ceiling* flags, and the *user list* (see Section 5.1.3).

New SCCS files created by the *admin* command are given mode 444 (read only). It is recommended that this mode *not* be changed, as it prevents any direct modification of the files by non-SCCS commands. It is further recommended that the directories containing SCCS files be given mode 755, which allows only the *owner* of the directory to modify its contents.

SCCS files should be kept in directories that contain only SCCS files and any temporary files created by SCCS commands. This simplifies protection and auditing of SCCS files (see Section 6.3). The contents of directories should correspond to convenient logical groupings, e.g., subsystems of a large project.

SCCS files must have only *one* link (name), because the commands that modify SCCS files do so by creating a copy of the file (the *x-file*, see Section 4) and, upon completion of processing, remove the old file and rename the *x-file*. If the old file has more than one link, this would break such additional links. Rather than process such files, SCCS commands produce an error message. All SCCS files *must* have names that begin with "s.".

When only one user uses SCCS, the real and effective user IDs are the same, and that user ID owns the directories containing SCCS files[10]. Therefore, SCCS may be used directly without any preliminary preparation.

However, in those situations in which several users with unique user IDs are assigned responsibility for one SCCS file (for example, in large software development projects), one user (equivalently, one user ID) must be chosen as the "owner" of the SCCS files and be the one who will "administer" them (e.g., by using the *admin* command). This user is termed the *SCCS administrator* for that project. Because other users of SCCS do not have the same privileges and permissions as the SCCS administrator, they are not able to execute directly those commands that require write permission in the directory containing the SCCS files. Therefore, a project-dependent program is required to provide an interface to the *get*, *delta*, and, if desired, *rmdel* and *cdc* commands.

The interface program must be owned by the SCCS administrator, and must have the *set user ID on execution* bit on (see *chmod*(1)), so that the effective user ID is the user ID of the administrator. This program invokes the desired SCCS command and causes it to *inherit* the privileges of the interface program for the duration of that command's execution. Thus, the owner of an SCCS file can modify it at will. Other users whose *login* names or *group* IDs are in the *user list* for that file (but who are *not* its owner) are given the necessary permissions only for the duration of the execution of the interface program, and are thus able to modify the SCCS files only through the use of *delta* and, possibly, *rmdel* and *cdc*. The project-dependent interface program, as its name implies, must be custom-built for each project.

---

10. Previously, the UNIX system allowed only 256 unique user IDs. Thus, several users often had to share user IDs and, therefore, file permissions. The current UNIX system allows 65,536 unique user IDs; it is recommended that each user have a unique user ID.

6.2 Format

SCCS files are composed of lines of ASCII text[11] arranged in six parts, as follows:

Checksum	A line containing the "logical" sum of all the characters of the file (*not* including this checksum itself).
Delta Table	Information about each delta, such as its type, its SID, date and time of creation, and commentary.
User Names	List of login names and/or group IDs of users who are allowed to modify the file by adding or removing deltas.
Flags	Indicators that control certain actions of various SCCS commands.
Descriptive Text	Arbitrary text provided by the user; usually a summary of the contents and purpose of the file.
Body	Actual text that is being administered by SCCS, intermixed with internal SCCS control lines.

Detailed information about the contents of the various sections of the file may be found in *sccsfile* (5); the *checksum* is the only portion of the file which is of interest below.

It is important to note that because SCCS files are ASCII files, they may be processed by various UNIX commands, such as *ed*(1), *grep*(1), and *cat*(1). This is very convenient in those instances in which an SCCS file must be modified manually (e.g., when the time and date of a delta was recorded incorrectly because the system clock was set incorrectly), or when it is desired to simply "look" at the file.

☞ *Extreme care should be exercised when modifying SCCS files with non-SCCS commands.*

6.3 Auditing

On rare occasions, perhaps due to an operating system or hardware malfunction, an SCCS file, or portions of it (i.e., one or more "blocks") can be destroyed. SCCS commands (like most UNIX commands) issue an error message when a file does not exist. In addition, SCCS commands use the *checksum* stored in the SCCS file to determine whether a file has been *corrupted* since it was last accessed (possibly by having lost one or more blocks, or by having been modified with, for example, *ed*(1)). *No* SCCS command will process a corrupted SCCS file except the *admin* command with the **−h** or **−z** keyletters, as described below.

It is recommended that SCCS files be audited (checked) for possible corruptions on a regular basis. The simplest and fastest way to perform an audit is to execute the *admin* command with the **−h** keyletter on all SCCS files:

```
admin −h s.file1 s.file2 ...
   or
admin −h directory1 directory2 ...
```

If the new checksum of any file is not equal to the checksum in the first line of that file, the message:

```
corrupted file (co6)
```

is produced for that file. This process continues until all the files have been examined. When examining directories (as in the second example above), the process just described will not

11. Previous versions of SCCS up to and including Version 3 used non-ASCII files. Therefore, files created by earlier versions of SCCS are incompatible with the current version of SCCS.

detect *missing* files. A simple way to detect whether *any* files are missing from a directory is to periodically execute the *ls*(1) command on that directory, and compare the outputs of the most current and the previous executions. Any file whose name appears in the previous output but not in the current one has been removed by some means.

Whenever a file has been corrupted, the manner in which the file is restored depends upon the extent of the corruption. If damage is extensive, the best solution is to contact the local UNIX operations group and request that the file be restored from a backup copy. In the case of minor damage, repair through use of the editor *ed*(1) may be possible. In the latter case, after such repair, the following command must be executed:

```
admin -z s.file
```

The purpose of this is to recompute the checksum to bring it into agreement with the actual contents of the file. After this command is executed on a file, any corruption which may have existed in that file will no longer be detectable.

January 1981

Function and Use of an SCCS Interface Program

L. E. Bonanni
A. Guyton (4/1/80 revision)

Bell Laboratories
Piscataway, New Jersey 08854

ABSTRACT

This memorandum discusses the use of a Source Code Control System Interface Program to allow more than one user to use SCCS commands upon the same set of files.

1. INTRODUCTION

In order to permit UNIX† users with different user identification numbers (user IDs) to use SCCS commands upon the same files, an SCCS interface program is provided to temporarily grant the necessary file access permissions to these users. This memorandum discusses the creation and use of such an interface program. This memorandum replaces an earlier version dated March 1, 1978.

2. FUNCTION

When only one user uses SCCS, the real and effective user IDs are the same, and that user ID owns the directories containing SCCS files. However, there are situations (for example, in large software development projects) in which it is practical to allow more than one user to make changes to the same set of SCCS files. In these cases, one user must be chosen as the **owner** of the SCCS files and be the one who will **administer** them (e.g., by using the *admin* command). This user is termed the *SCCS administrator* for that project. Since other users of SCCS do not have the same privileges and permissions as the SCCS administrator, they are not able to execute directly those commands that require write permission in the directory containing the SCCS files. Therefore, a project-dependent program is required to provide an interface to the *get*, *delta*, and, if desired, *rmdel*, *cdc*, and *unget* commands.[1]

The interface program must be owned by the SCCS administrator, must be executable by non-owners, and must have the *set user ID on execution* bit on (see *chmod*(1)[2]), so that, when executed, the *effective* user ID is the user ID of the administrator. This program's function is to invoke the desired SCCS command and to cause it to *inherit* the privileges of the SCCS administrator for the duration of that command's execution. In this manner, the owner of an SCCS file (the administrator) can modify it at will. Other users whose *login* names are in the *user list*[3] for that file (but who are *not* its owners) are given the necessary permissions only for the duration of the execution of the interface program, and are thus able to modify the SCCS files only through the use of *delta* and, possibly, *rmdel* and *cdc*.

---

† UNIX is a trademark of Bell Laboratories.

1. Other SCCS commands either do not require write permission in the directory containing SCCS files or are (generally) reserved for use only by the administrator.
2. All references of the form *name*(*N*) refer to item *name* in section *N* of the *UNIX User's Manual*.
3. This is the list of login names of users who are allowed to modify an SCCS file by adding or removing deltas. The login names are specified using the *admin*(1) command.

3. A BASIC PROGRAM

When a UNIX program is executed it is passed (as argument 0) the *name* by which it is invoked, followed by any additional user-supplied arguments. Thus, if a program is given a number of *links* (names), it may alter its processing depending upon which link is used to invoke it. This mechanism is used by an SCCS interface program to determine which SCCS command it should subsequently invoke (see *exec*(2)).

A generic interface program (**inter.c**, written in C) is shown in *Attachment I*. Note the reference to the (unsupplied) function **filearg**. This is intended to demonstrate that the interface program may also be used as a pre-processor to SCCS commands. For example, function **filearg** could be used to modify file arguments to be passed to the SCCS command by supplying the *full* path name of a file, thus avoiding extraneous typing by the user. Also, the program could supply any additional (default) keyletter arguments desired.

4. LINKING AND USE

In general, the following demonstrates the steps to be performed by the SCCS administrator to create the SCCS interface program. It is assumed, for the purposes of the discussion, that the interface program **inter.c** resides in directory **/x1/xyz/sccs**. Thus, the command sequence:

```
cd /x1/xyz/sccs
cc ... inter.c -o inter ...
```

compiles **inter.c** to produce the executable module **inter** (... represents arguments that may also be required). The proper mode and the *set user ID on execution* bit are set by executing:

```
chmod 4755 inter
```

Finally, new links are created, by (for example):[4]

```
ln inter get
ln inter delta
ln inter rmdel
```

Subsequently, *any* user whose shell parameter PATH (see *sh*(1)) specifies that directory **/x1/xyz/sccs** is to be searched first for executable commands, may execute, for example:

```
get -e /x1/xyz/sccs/s.abc
```

from any directory to invoke the interface program (via its link *get*). The interface program then executes **/usr/bin/get** (the actual SCCS *get* command) upon the named file. As previously mentioned, the interface program could be used to supply the pathname **/x1/xyz/sccs**, so that the user would only have to specify:

```
get -e s.abc
```

to achieve the same results.

5. CONCLUSION

An SCCS interface program is used to permit users having different user IDs to use SCCS commands upon the same files. Although this is its primary purpose, such a program may also be used as a pre-processor to SCCS commands since it can perform operations upon its arguments.

---

4. The names of the links may be arbitrary, provided the interface program is able to determine from them the names of SCCS commands to be invoked.

Attachment I

SCCS Interface Program **inter.c**

```
main(argc, argv)
int argc;
char *argv[];
{
        register int i;
        char cmdstr[LENGTH]

        /*
        Process file arguments (those that don't begin with "-").
        */
        for (i = 1; i < argc; i++)
                if (argv[i][0] != '-')
                        argv[i] = filearg(argv[i]);

        /*
        Get "simple name" of name used to invoke this program
        (i.e., strip off directory-name prefix, if any).
        */
        argv[0] = sname(argv[0]);

        /*
        Invoke actual SCCS command, passing arguments.
        */
        sprintf(cmdstr, "/usr/bin/%s", argv[0]);
        execv(cmdstr, argv);
}
```

January 1981

BC—An Arbitrary Precision Desk-Calculator Language

Lorinda Cherry

Robert Morris

Bell Laboratories
Murray Hill, New Jersey 07974

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.

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.

† UNIX is a trademark of Bell Laboratories.

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 content of 'obase', initially 10, is 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] *UNIX Programmer's Manual*, Bell Laboratories.

[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, 1975.

[4] S. C. Johnson, *YACC—Yet Another Compiler-Compiler*, Bell Laboratories.

[5] R. Morris and L. L. Cherry, *DC—An Interactive Desk Calculator*, Bell Laboratories.

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. New-line 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 maximum 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 new-line. 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 new-line 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.

DC—An Interactive Desk Calculator

Robert Morris

Lorinda Cherry

Bell Laboratories
Murray Hill, New Jersey 07974

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.

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:

† UNIX is a trademark of Bell Laboratories.

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

$$x_{n+1} = \tfrac{1}{2}(x_n + \frac{y}{x_n})$$

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 and R. Morris. *BC—An Arbitrary Precision Desk-Calculator Language*, Bell Laboratories.

[2] K. C. Knowlton. A Fast Storage Allocator, *CACM* **8**(10):623-25 (Oct. 1965).

January 1981

UNIX Graphics Overview

A. R. Feuer

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

UNIX† Graphics, or just *graphics*, is the name given to a growing collection of numerical and graphical commands available as part of UNIX [1]. In its initial release, *graphics* includes commands to construct and edit numerical data plots and hierarchy charts. This memorandum will help you get started using *graphics* and show you where to find more information. The examples below assume that you are familiar with the UNIX shell [1].

2. BASIC CONCEPTS

The basic approach taken in *graphics* is to generate a drawing by describing it rather than by drafting it. Any drawing is seen as having two fundamental attributes: its underlying logic and its visual layout. The layout encompasses one representation of the logic. For example, consider the attributes of a drawing that consists of a plot of the function $y=x^2$ for x between 0 and 10. The logic of the plot is the description as just given, viz. $y=x^2, 0 \leq x \leq 10$. The layout consists of an x-y grid, axes labeled perhaps 0 to 10 and 0 to 100, and lines drawn connecting the x-y pairs 0,0 to 1,1 to 2,4 and so on.

The way to generate a picture in *graphics* is

gather data | transform the data | generate a layout | display the layout.

To generate the specific plot of $y=x^2, 0 \leq x \leq 10$ and display it on a Tektronix display terminal would be

```
gas -s0,t10 | af "x^2" | plot | td
```

where

gas generates sequences of numbers, in this case starting at 0 and terminating at 10.

af performs general arithmetic transformations.

plot builds x-y plots.

td displays drawings on Tektronix terminals.

The resulting drawing is shown in Figure 1.

The layout generated by a *graphics* program may not always be precisely what is wanted. There are two ways to influence the layout. Each drawing program accepts options to direct certain layout features. For instance, in the previous example we may have wanted the x-axis labels to indicate each of the numbers plotted and we might not have wanted any y-axis labels at all. To achieve this the *plot* command would be changed to:

```
plot -xi1,ya
```

producing the drawing of Figure 2.

---

† UNIX is a trademark of Bell Laboratories.

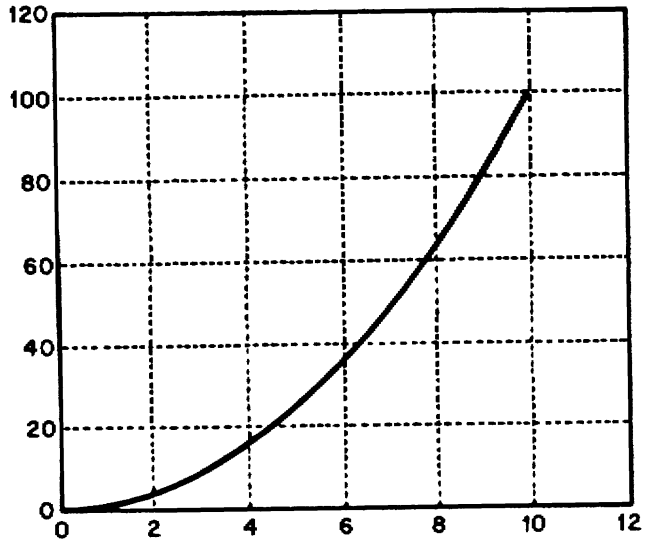


Figure 1. gas -s0,t10 I af "x^2" I plot I td

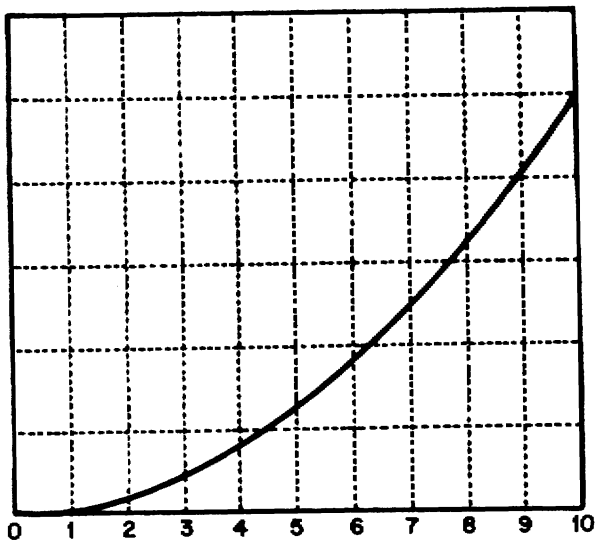


Figure 2. gas -s0,t10 I af "x^2" I plot -x11,ya I td

The output from any drawing command can also be affected by editing it directly at a display terminal using the graphical editor, *ged*. To edit a drawing really means to edit the computer representation of the drawing. In the case of *graphics* the representation is called a graphical primitive string, or GPS. All of the drawing commands (e.g., *plot*) write GPS and all of the device filters (e.g., *td*) read GPS. *Ged* allows you to manipulate GPS at a display terminal by interacting with the drawing the GPS describes.

GPS describes graphical objects drawn within a Cartesian plane 65,534 units on each axis. The plane, known as the *universe*, is partitioned into 25 equal sized square regions. Multi-drawing displays can be produced by placing drawings into adjacent regions and then displaying each region.

3. GETTING STARTED

To access the *graphics* commands when logged in on a UNIX system type **graphics**. Your *shell* variable **PATH** will be altered to include the *graphics* commands and the *shell* primary prompt will be changed to ". Any command accessible before typing **graphics** will still be accessible; *graphics* only adds commands, it doesn't take any away. Once in *graphics*, you can find out about any of the *graphics* commands using *whatis*. Typing **whatis** by itself on a command line will generate a list of all the commands in *graphics* along with instructions on how to find out more about any of them.

All of the *graphics* commands accept the same command line format:

A *command* is:	a *command-name* followed by *argument*(s).
A *command-name* is:	the name of any of the *graphics* commands.
An *argument* is:	a *file-name* or an *option-string*.
A *file-name* is:	any file name not beginning with −, or a − by itself to reference the standard input.
An *option-string* is:	a − followed by *option*(s).
An *option* is:	letter(s) followed by an optional value. Options may be separated by commas.

You will get the best results with *graphics* commands if you use a display terminal. *Tplot*(1G) filters can be used in conjunction with *gtop* (see *gutil*(1G)) to get somewhat degraded drawings on Versatec printers and Dasi-type terminals. And since GPS can be stored in a file, it can be created from any terminal for later displaying on a graphical device.

To remove the *graphics* commands from your **PATH** *shell* variable type **EOT** (control-d on most terminals). To log off UNIX from *graphics* type **quit**.

4. EXAMPLES OF WHAT YOU CAN DO

4.1 Numerical Manipulation and Plotting

Stat(1G) describes a collection of numerical commands. All of these commands operate on vectors. A vector is a text file that contains numbers separated by delimiters, where a delimiter is anything that is not a number. For example:

```
1 2 3 4 5, and
arf tty47 Mar 5 09:52
```

are both vectors. (The latter being the vector: 47 5 9 52.)

Here is an easy way to generate a Celsius-Fahrenheit conversion table using *gas* to generate the vector of Celsius values:

```
gas −s0,t100,i10 | af "C,9/5*C+32"
```

The output is:

```
0.0     32
10      50
20      68
30      86
40      104
50      122
60      140
70      158
80      176
90      194
100     212
```

This is what is going on:

gas −s0,t100,i10 — We have seen *gas* in an earlier example. In this case the sequence starts at 0, terminates at 100, and the increment between successive elements is 10.

af "C,9/5*C+32" — We have also seen *af*. Arguments to *af* are expressions. Operands in an expression are either constants or file names. If a file name is given that does not exist in the current directory it is taken as the name for the standard input. In this example **C** references the standard input. The output is a vector with odd elements coming from the standard input and even elements being a function of the preceding odd element.

Here is an example that illustrates the use of vector titles and multiline plots:

```
gas | title −v"first ten integers" >N
root N >RN
root −r3 N >R3N
root −r1.5 N >R1.5N
plot −FN,g N R1.5N RN R3N | td
```

The resulting plot is shown in Figure 3.

title −v"*name*" — *Title* associates a *name* with a vector. In this case, **first ten integers** is associated with the vector output by *gas*. The vector is stored in file **N**.

root −r*n* — *Root* outputs the *n*th root of each element on the input. If **−r*n*** is not given then the square root is output. Also, if the input is a titled vector the title will be transformed to reflect the root function.

plot −F*X,g Y(s)* — This command generates a multiline plot with *Y(s)* plotted versus *X*. The **g** option causes tick marks to appear instead of grid lines.

The next example generates a histogram of random numbers:

```
rand −n100 | title −v"100 random numbers" | qsort | bucket | hist | td
```

The output is shown in Figure 4.

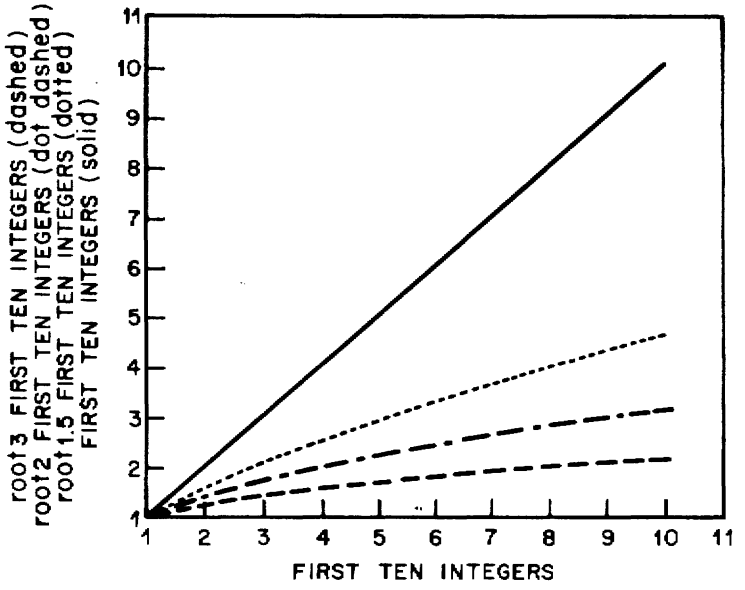


Figure 3. Some roots of the first ten integers

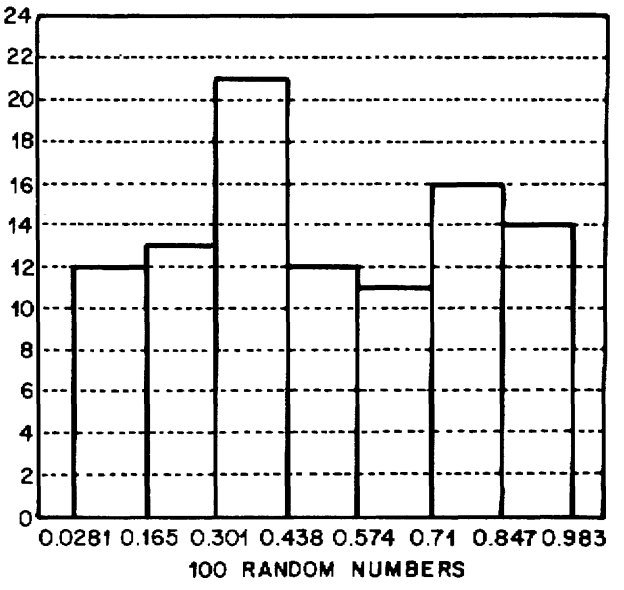


Figure 4. Histogram of 100 random numbers

rand −n100	*Rand* outputs random numbers using *rand*(3C). In this case 100 numbers are output in the range 0 to 1.
qsort	*Qsort* sorts the elements of a vector in ascending order.
bucket	*Bucket* breaks the range of a vector into intervals and counts how many elements from the vector fall into each interval. The output is a vector with odd elements being the interval boundaries and even elements being the counts.
hist	*Hist* builds a histogram based on interval boundaries and counts.

4.2 Drawings Built from Boxes

There is a large class of drawings composed from boxes and text. Examples are structure charts, configuration drawings, and flow diagrams. In *graphics* the general procedure to construct such box drawings is the same as that for numerical plotting. Namely gather and transform the data, build and display the layout.

As an example, consider hierarchy charts. The command line:

dtoc | vtoc | td

outputs the drawing shown in Figure 5.

Dtoc outputs a table of contents that describes a directory structure (Figure 5a). The fields from left to right are level number, directory name, and the number of ordinary readable files contained in the directory. *Vtoc* reads a (textual) table of contents and outputs a visual table of contents, or hierarchy chart. Input to *vtoc* consists of a sequence of entries, each describing a box to be drawn. An entry consists of a level number, an optional style field, a text string to be placed in the box, and a mark field to appear above the top right hand corner of the box.

5. WHERE TO GO FROM HERE

The best way to learn about *graphics* is to log onto a UNIX system and use it. Tutorials exist for *stat*(1G) [3] and *ged*(1G) [4]; [2] contains administrative information for *graphics*. Reference information can be found in the *UNIX User's Manual* in the following manual entries:

gdev(1G), a collection of commands to manipulate Tektronix 4000 series terminals; and
ged(1G), the graphical editor;
graphics(1G), the entry point for *graphics*;
gutil(1G), a collection of utility commands;
stat(1G), numerical manipulation and plotting commands;
toc(1G), routines to build tables of contents;
gps(5), a description of a graphical primitive string.

6. REFERENCES

[1] T. A. Dolotta, S. B. Olsson, and A. G. Petruccelli (eds.). *UNIX User's Manual*—Release 3.0, Bell Laboratories (June 1980).

[2] R. L. Chen, D. E. Pinkston, and A. Guyton. *Administrative Information for the UNIX Graphics Package*, Bell Laboratories.

[3] A. R. Feuer and A. Guyton. *STAT—A Tool for Analyzing Data*, Bell Laboratories.

[4] A. R. Feuer. *A Tutorial Introduction to the Graphics Editor*, Bell Laboratories.

Figure 5. Directory Structure for Graphics

```
0.      "source"        2
1.      "glib.d"        1
1.1.    "gpl.d"         12
1.2.    "gsl.d"         14
2.      "gutil.d"       6
2.1.    "cvrtopt.d"     7
2.2.    "gtop.d"        8
2.3.    "ptog.d"        5
3.      "stat.d"        54
4.      "tek4000.d"     5
4.1     "ged.d"         37
4.4.    "td.d"          8
5.      "toc.d"         3
5.1.    "ttoc.d"        3
5.2.    "vtoc.d"        22
6.      "whatis.d"      108
```

Figure 5a. Dtoc output

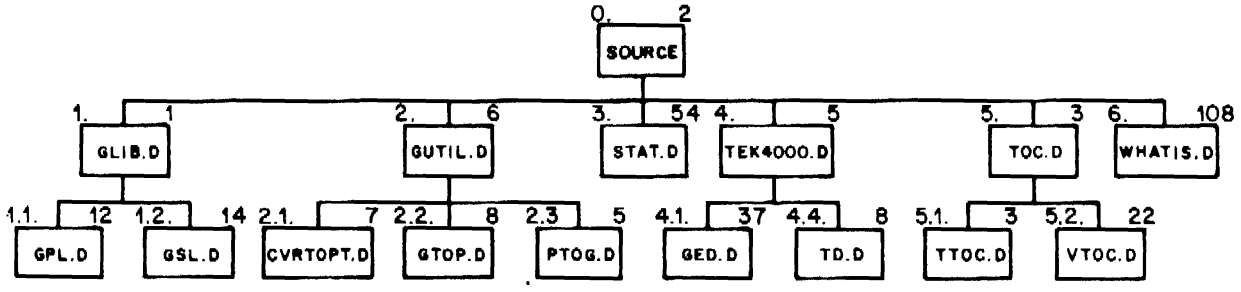


Figure 5b. Vtoc output

A Tutorial Introduction to the Graphics Editor

A. R. Feuer

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

Ged is an interactive graphical editor used to display, edit, and construct drawings on Tektronix 4010 series display terminals. The drawings are represented as a sequence of objects in a token language known as GPS (for graphical primitive string). GPS is produced by the drawing commands in UNIX† Graphics [1] such as *vtoc* and *plot*, as well as by *ged* itself.

The examples in this tutorial illustrate how to construct and edit simple drawings. Try them to become familiar with how the editor works, but keep in mind that *ged* is intended primarily to edit the output of other programs rather than to construct drawings from scratch. A summary of editor commands and options is given in Section 3.

As for notation, literal keystrokes are printed in **boldface**. Meta-characters are also in boldface and are surrounded by angled brackets. For example, **<cr>** means return and **<sp>** means space. In the examples, output from the terminal is printed in roman (normal) type. In-line comments are in roman and are surrounded by parentheses.

2. COMMANDS

To start we will assume that you have successfully entered the graphics environment (as described in *graphics*(1G) of [2]) while logged in at a display terminal. To enter *ged* type:

ged <cr>

After a moment the screen should be clear save for the *ged* prompt, *, in the upper left corner. The * tells you that *ged* is ready to accept a command.

Each command passes through a sequence of stages during which you describe what the command is to do. All commands pass through a subset of these stages:

1. *command line*
2. *text*
3. *points*
4. *pivot*
5. *destination*

As a rule, each stage is terminated by typing **<cr>**. The **<cr>** for the last stage of a command triggers execution.

2.1 The Command Line

The simplest commands consist only of a *command line*. The *command line* is modeled after a conventional command line in the shell. That is:

command-name [−*option*(s)] [*filename*] **<cr>**

? is an example of a simple command. It lists the commands and options understood by *ged*. To generate the list, type:

***? <cr>** (you type a question mark followed by a return)

† UNIX is a trademark of Bell Laboratories.

A command is executed by typing the first character of its name. *Ged* will echo the full name and wait for the rest of the *command line*. For example, e references the *erase* command. As *erase* consists only of stage 1, typing <cr> causes the erase action to occur. Typing <**rubout**> after a command name and before the final <cr> for the command aborts the command. Thus while

*erase <cr>

erases the display screen,

*erase <**rubout**>

brings the editor back to *.

Following the command-name, *options* may be entered. Options control such things as the width and style of lines to be drawn or the size and orientation of text. Most options have a default value that applies if a value for the option is not specified on the command line. The *set* command allows you to examine and modify the default values. Type:

*set <cr>

to see the current default values.

The value of an option is either of type integer, character, or Boolean. Boolean values are represented by + for true and − for false. A default value is modified by providing it as an option to the *set* command. For example, to change the default text height to 300 units type:

*set −**h300** <cr>

Arguments on the command line, but not the command-name, may be edited using the erase and kill characters from the shell. (Actually, this applies whenever text is being entered.)

2.2 Constructing Graphical Objects

Drawings are stored as GPS in a *display buffer* internal to the editor. Typically, a drawing in *ged* is composed of instances of three graphical primitives: *arc*, *lines*, and *text*.

2.2.1 Generating Text. To put a line of text on the display screen use the *Text* command. First enter the *command line* (stage 1):

*Text <cr>

Next enter the *text* (stage 2):

a line of text <cr>

And then enter the starting *point* for the text (stage 3):

<**position cursor**> <cr>

Positioning of the graphic cursor is done either with the thumbwheel knobs on the terminal keyboard or with an auxiliary joystick. The <cr> establishes the location of the cursor to be the starting point for the text string. The *Text* command ends at stage 3, so this <cr> initiates the drawing of the text string.

Text accepts options to vary the angle, height, and line width of the characters, and to either center or right justify the text object. The text string may span more than one line by escaping the <cr> (i.e., \<cr>) to indicate continuation. To illustrate some of these capabilities, try the following:

```
*Text -r <cr>                          (right justify text)
top\<cr>
right <cr>
<position cursor> <cr>
*Text -a90 <cr>                        (rotate text 90 degrees)
lower\<cr>
left <cr>
<position cursor> <cr>                 (pick a point below and left of the previous point)
```

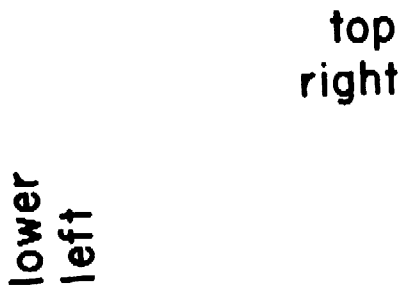


Figure 1. Generating text objects

2.2.2 Drawing Lines. The *Lines* command is used to construct objects built from a sequence of straight lines. It consists of stages 1 and 3. Stage 1 is straightforward:

*Lines *possible options* <cr>

Lines accepts options to specify line style and line width.

Stage 3, the entering of *points*, is more interesting. *Points* are referenced either with the graphic cursor or by name. We have already entered a point with the cursor for the *Text* command. For *Lines* it is more of the same. As an example, let us build a triangle:

```
*Lines <cr>
<position cursor> <sp>          (locate the first point)
<position cursor> <sp>          (the second point)
<position cursor> <sp>          (the third point)
<position cursor> <sp>          (back to the first point)
<cr>                            (terminate points, draw triangle)
```

Typing <sp> enters the location of the crosshairs as a point. *Ged* identifies the point with an integer and adds the location to the current *point set*. The last point entered can be erased by typing # . The current point set can be cleared by typing @ . On receiving the final <cr> the points are connected in numerical order.

The points in the current point set may be referenced by name using the $ operator. $*n* references the point numbered *n*. Using $ we can redraw the triangle above by entering:

```
*Lines <cr>
<position cursor> <sp>
<position cursor> <sp>
<position cursor> <sp>
$0 <cr>                         (reference point 0)
<cr>
```

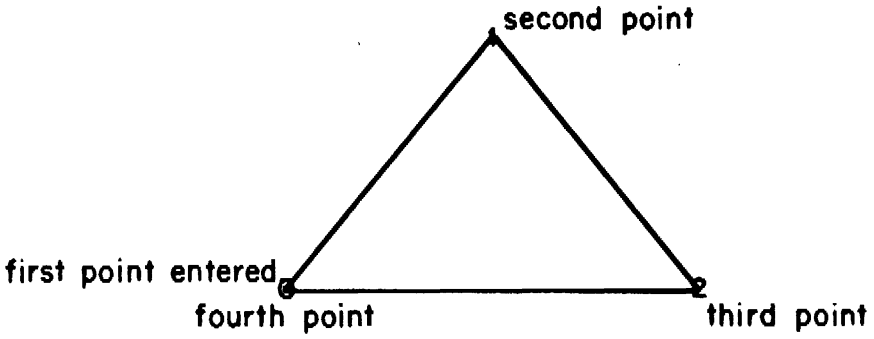


Figure 2. Building a triangle

At the start of each command that includes stage 3, *points*, the current point set is empty. The point set from the previous command is saved and is accessible using the . operator; . swaps the points in the previous point set with those in the current set. The = operator can be used to identify the current points. To illustrate, let us use the triangle just entered as the basis for drawing a quadrilateral:

***Lines <cr>**	
.	(access the previous point set)
=	(identify the current points)
#	(erase the last point)
<position cursor> <sp>	(add a new point)
$0 <cr>	(close the figure)
<cr>	

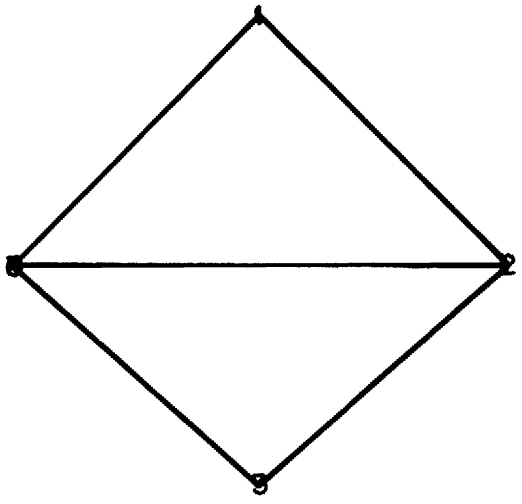


Figure 3. Accessing the previous point set

Individual points from the previous point set can be referenced by using the . operator with **$**. We will build a triangle that shares an edge with the quadrilateral:

```
*Lines <cr>
$.1 <cr>          (reference point 1 from the previous point set)
$.2 <cr>          (reference point 2)
<sp>              (enter a new point)
$0 <cr>           (or $.1, to close the figure)
<cr>
```

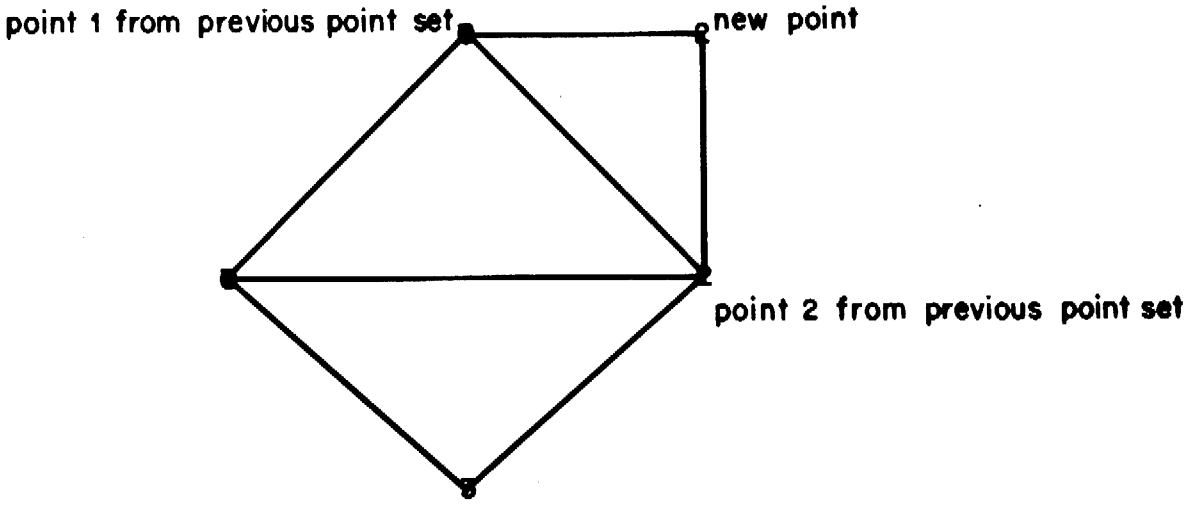


Figure 4. Referencing points from the previous point set

A point can also be given a name. The > operator allows you to associate an upper case letter with a point just entered. A simple example is:

```
*Lines <cr>
<position cursor> <sp>      (enter a point)
>A                          (name the point A)
<position cursor> <sp>
<cr>
```

In commands that follow you can now reference point A using the $ operator, as in:

```
*Lines <cr>
$A
<position cursor> <sp>
<cr>
```

2.2.3 Drawing Curves. Curves are interpolated from a sequence of three or more points. The *Arc* command generates a circular arc given three points on a circle. The arc is drawn starting at the first point, through the second point, and ending at the third point. A circle is an arc with the first and third points coincident. One way to draw a circle is thus:

```
*Arc <cr>
<position cursor> <sp>
<position cursor> <sp>
$0 <cr>
<cr>
```

2.3 Editing Objects

2.3.1 Addressing Objects. An object is addressed by pointing to one of its *handles*. All objects have an *object-handle*. Usually the object-handle is the first point entered when the object was created. The *objects* command marks the location of each object-handle with an **O**. Type:

***objects −v <cr>**

to see the handles of all the objects on the screen.

Some objects, *Lines* for example, also have *point-handles*. Typically each of the points entered when an object is constructed becomes a point-handle. (Yes, an object-handle is also a point-handle.) The *points* command marks each of the point-handles.

A handle is pointed to by including it within a *defined-area*. A defined-area is generated either with a command line option or interactively using the graphic cursor. As an example, try deleting one of the objects you have created on the screen.

***Delete <cr>**	
<position cursor> <sp>	(above and to the left of some object-handle)
<position cursor> <sp>	(below and to the right of the object-handle)
<cr>	(the defined-area should include the object-handle)
<cr>	(if all is well, delete the object)

The defined-area is outlined with dotted lines. The reason for the seemingly extra **<cr>** at the end of the *Delete* command is to give you an opportunity to stop the command (using **<rubout>**) if the defined-area is not quite right. Every command that accepts a defined-area will wait for a confirming **<cr>** . Use the *new* command to get a fresh copy of the remaining objects.

Notice that defined-areas are entered as *points* in the same way that objects are created. Actually, a defined-area may be generated by giving anywhere from zero to 30 points. Inputting zero points is particularly useful to point to a single handle. It creates a small defined-area about the location of the terminating **<cr>** . Using a zero point defined-area, the *Delete* command would be:

***Delete <cr>**	
<position cursor>	(center the crosshairs on the object-handle)
<cr>	(terminate the defined-area)
<cr>	(delete the object)

A defined-area can also be given as a command line option. For example, to delete everything in the display buffer give the **u**niverse option to the *Delete* command. Note the difference between the commands **Delete −u** and **erase**.

2.3.2 Changing the Location of an Object. Objects are moved using the *Move* command. Create a circle using *Arc*, then move it as follows:

***Move <cr>**	
<position cursor> <cr>	(centered on the object-handle)
<cr>	(this establishes a *pivot*, marked with an asterisk)
<position cursor> <cr>	(this establishes a *destination*)

The basic move operation relocates every point in each object addressed by the distance from the *pivot* to the *destination*. In this case we chose the pivot to be the object-handle, so effectively we moved the object-handle to the destination point.

2.3.3 Changing the Shape of an Object. The *Box* command is a special case of generating lines. Given two points it creates a rectangle such that the two points are at opposite corners. The sides of the rectangle lie parallel to the edges of the screen. Draw a box:

```
*Box <cr>
<position cursor> <sp>
<position cursor> <cr>
```

Box generates point-handles at each vertex of the rectangle. Use the *points* command to mark the point-handles. The shape of an object can be altered by moving point-handles. The next example illustrates one way to double the height of a box.

```
*Move -p+ <cr>
<position cursor> <sp>          (left of the box, between the top and bottom edges)
<position cursor> <cr>          (right of the box, below the bottom edge)
<position cursor> <cr>          (on the top edge)
<position cursor> <cr>          (directly below on the bottom edge)
```

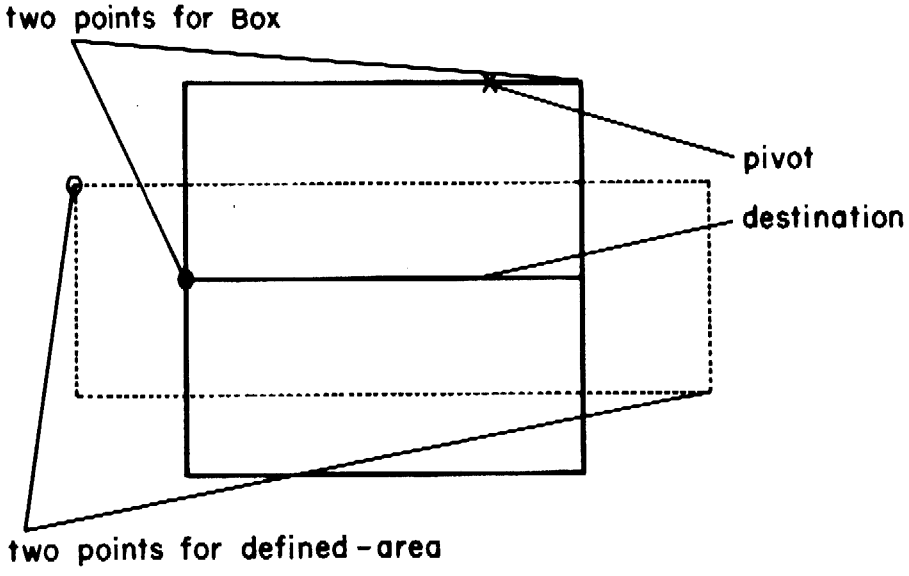


Figure 5. Growing a box

Because the points flag is true, the operation is applied to each point-handle addressed. In this case each point-handle within the defined-area is moved the distance from the pivot to the destination. If **p** were false only the object-handle would have been addressed.

2.3.4 Changing the Size of an Object. The size of an object can be changed using the *Scale* command. *Scale* scales objects by changing the distance from each handle of the object to a pivot by a factor. Put a line of text on the screen and try the following *Scale* commands:

```
*Scale -f200 <cr>               (factor is in percent)
<position cursor> <cr>          (point to object-handle)
<position cursor> <cr>          (set pivot to rightmost character)
<cr>

*Scale -f50 <cr>
. <cr>                          (reference the previous defined-area)
<position cursor> <cr>          (set pivot above a character near the middle)
<cr>
```

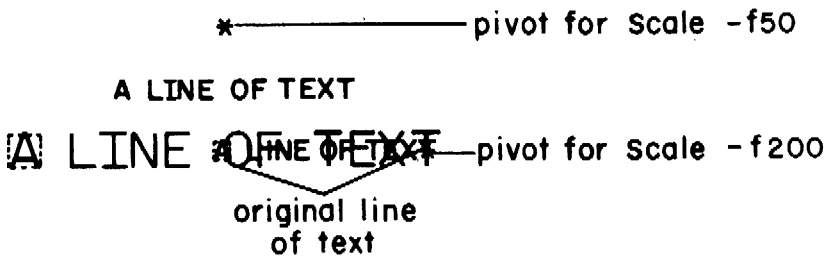


Figure 6. Scaling text

A useful insight into the behavior of scaling is to note that the position of the pivot does not change. Also observe that the defined-area is scaled to preserve its relationship to the graphical objects.

The size of objects can also be changed by moving point-handles. Generate a circle, this time using the *Circle* command:

***Circle <cr>**	
<position cursor> <sp>	(specify the center)
<position cursor> <cr>	(specify a point on the circle)

Circle generates an arc with the first and third point at the point specified on the circle. The second point of the arc is located 180° around the circle. One way to change the size of the circle is to move one of the point-handles (using Move **−p**).

The size of text characters can be changed via a third mechanism. Character height is a property of a line of text. The *Edit* command allows you to change character height as follows:

***Edit −h*height* <cr>**	(*height* is in universe units, see Section 2.4)
<position cursor> <cr>	(point to the object-handle)
<cr>	

2.3.5 Changing the Orientation of an Object. The orientation of an object can be altered using *Rotate*. *Rotate* rotates each point of an object about a pivot by an angle. Try the following rotations on a line of text:

***Rotate −a90 <cr>**	(angle is in degrees)
<position cursor> <cr>	(point to object-handle)
<position cursor> <cr>	(set pivot to rightmost character)
<cr>	
***Rotate −a−90 <cr>**	
. <cr>	(reference previous defined-area)
<position cursor> <cr>	(set pivot to a character near the middle)
<cr>	

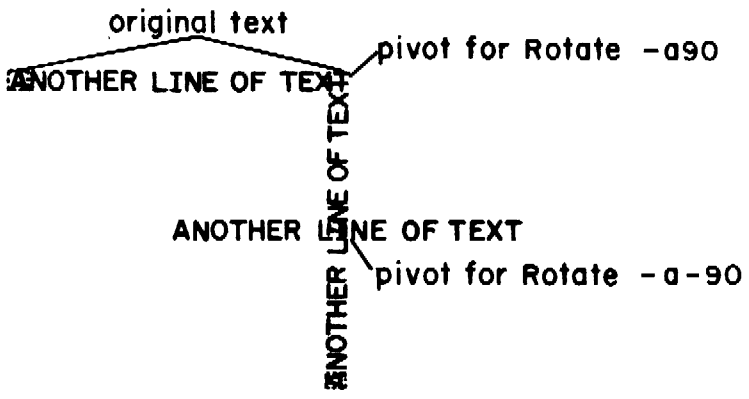


Figure 7. Rotating text

2.3.6 Changing the Style or Width of Lines. In the current editor objects can be drawn from lines in any of five styles (**s**olid, **d**ashed, **d**ot-dashed, **d**otted, long-dashed) and three widths (**n**arrow, **m**edium, **b**old). Style is controlled by the **s** option, width by **w**:

```
*Lines −wn,sdo <cr>
<position cursor> <sp>
<position cursor> <sp>
<cr>
```

creates a narrow width dotted line.

```
*Edit −wb,sdd <cr>
<position cursor> <cr>          (point to object-handle of the line)
<cr>
```

changes the line to bold dot-dashed.

2.4 View Commands

All of the objects we have drawn lie within a Cartesian plane, 65,534 units on each axis, known as the *universe*. Thus far we have displayed only a small portion of the universe on the display screen. The command:

```
*view −u <cr>
```

displays the entire universe.

A mapping of a portion of the universe onto the display screen is called a *window*. The extent or magnification of a window is altered using the *zoom* command. To build a window that includes all of the objects you have drawn, type:

```
*zoom <cr>
<position cursor> <sp>          (above and to the left of any object)
<position cursor> <cr>          (below and to the right, also end points)
<cr>                            (verify)
```

Zooming can be either *in* or *out*. Zooming in, as with a camera lens, increases the magnification of the window. The area outlined by *points* is expanded to fill the screen. Zooming out decreases magnification. The current window is shrunk so that it fits within the defined-area. The direction of the zoom is controlled by the sense of the **o**ut flag; **o** true means zoom out.

The location of a window is altered using *view*. *View* moves the window so that a given point in the universe lies at a given location on the screen.

```
*view <cr>
<position cursor> <cr>          (locate a point in the universe)
<position cursor> <cr>          (locate a point on the screen)
```

View also provides access to several predefined windows. We have already seen view **−u**. view **−h** displays the *home-window* . The home-window is the window that circumscribes all of the objects in the universe. The result is similar to that of the example using *zoom* given earlier.

Lastly, using *view* you may select to window on a particular *region*. The universe is partitioned into 25 equal sized regions. Regions are numbered from 1 to 25 beginning at the lower left and proceeding toward the upper right. Region 13, the center of the universe, is used as the default region by drawing commands such as *plot* and *vtoc* (see [1]).

2.5 Other Commands

2.5.1 Interacting with Files. To save the contents of the display buffer copy it to a file using the *write* command:

*write *filename* <cr>

The contents of *filename* will be a GPS, thus it can be displayed using any of the device filters (e.g., *td* [1]) or read back into *ged*.

A GPS is read into the editor using the *read* command:

*read *filename* <cr>

The GPS from *filename* is appended to the display buffer and then displayed. Because *read* does not change the current window, only some (or none) of the objects read may be visible. A useful command sequence to view everything read is:

```
*read −e− filename <cr>
*view −h <cr>
```

The display function of *read* is inhibited by setting the echo flag to false; view **−h** windows on and displays the full display buffer.

The *read* command may also be used to input text files. The form is:

read [−*option*(s)] *filename* <cr>

followed by a single point to locate the first line of text. A text object is created for each line of text from *filename*. Options to *read* are the same as those for the *Text* command.

2.5.2 Leaving the Editor. Use the *quit* command to terminate an editing session. As with the text editor *ed*, *quit* responds with **?** if the internal buffer has been modified since the last *write*. A second *quit* forces exit.

2.6 Other Useful Things to Know

2.6.1 One-Line UNIX Escape. As in *ed*, **!** provides a temporary escape to the shell.

2.6.2 Typing Ahead. Most programs under UNIX allow you to type input before the program is ready to receive it. In general, this is not the case with *ged*; characters typed before the appropriate prompt are lost.

2.6.3 Speeding up Things. Displaying the contents of the display buffer can be time consuming, particularly if much text is involved. The wise use of two flags to control what gets displayed can make life more pleasant: the echo flag controls echoing of new additions to the display buffer; the text flag controls whether text will be outlined or drawn.

3. COMMAND SUMMARY

In the summary, characters actually typed are printed in boldface. Command stages are printed in italics. Arguments surrounded by brackets are optional. Parentheses surrounding arguments separated by "or" means that exactly one of the arguments must be given. For example, the *Delete* command (Section 3.2) accepts the arguments **−universe**, **−view**, and *points*.

3.1 Construct commands:

Arc	[**−echo,style,width**] *points*
Box	[**−echo,style,width**] *points*
Circle	[**−echo,style,width**] *points*
Hardware	[**−echo**] *text points*
Lines	[**−echo,style,width**] *points*
Text	[**−angle,echo,height,midpoint,rightpoint,text,width**] *text points*

3.2 Edit commands:

Delete	(**−** (**universe** or **view**) or *points*)
Edit	[**−angle,echo,height,style,width**] (**−** (**universe** or **view**) or *points*)
Kopy	[**−echo,points,x**] *points pivot destination*
Move	[**−echo,points,x**] *points pivot destination*
Rotate	[**−angle,echo,kopy,x**] *points pivot destination*
Scale	[**−echo,factor,kopy,x**] *points pivot destination*

3.3 View commands:

coordinates	*points*
erase	
new	
objects	(**−** (**universe** or **view**) or *points*)
points	(**−** (**labelled-points** or **universe** or **view**) or *points*)
view	(**−** (**home** or **universe** or **region**) or [**−x**] *pivot destination*)
x	[**−view**] *points*
zoom	[**−out**] *points*

3.4 Other commands:

quit	
read	[**−angle,echo,height,midpoint,rightpoint,text,width**] *filename* [*destination*]
set	[**−angle,echo,factor,height,kopy,midpoint,points,rightpoint,style,text,width,x**]
write	*filename*
!*command*	
?	

3.5 Options:

Options specify parameters used to construct, edit, and view graphical objects. If a parameter used by a command is not specified as an *option*, the default value for the parameter will be used. The format of command *options* is:

−*option* [,*option*]

where *option* is *keyletter*[*value*]. Flags take on the *values* of true or false indicated by + and − respectively. If no *value* is given with a flag, true is assumed.

Object Options:

angle*n*	Specify an angle of *n* degrees.
echo	When true, changes to the display buffer will be echoed on the screen.
factor*n*	Specify a scale factor of *n* percent.
height*n*	Specify height of *text* to be *n* universe-units ($0 \leq n < 1280$).
kopy	The commands *Scale* and *Rotate* can be used to either create new objects or to alter old ones. When the **kopy** flag is true, new objects are created.
midpoint	When true, use the midpoint of a text string to locate the string.
out	When true, reduce magnification during *zoom*.
points	When true, operate on points otherwise operate on objects.
rightpoint	When true, use the rightmost point of a text string to locate the string.
style*type*	Specify line style to be one of following *types*: **so** solid **da** dashed **dd** dot-dashed **do** dotted **ld** long-dashed
text	Most text is drawn as a sequence of lines. This can sometimes be painfully slow. When the **text** flag is false, *text* strings are outlined rather than drawn.
width*type*	Specify line width to be one of following *types*: **n** narrow **m** medium **b** bold
x	One way to find the center of a rectangular area is to draw the diagonals of the rectangle. When the **x** flag is true, defined-areas are drawn with their diagonals.

Area Options:

home	Reference the home-window.
region*n*	Reference region *n*.
universe	Reference the universe-window.
view	Reference those objects currently in view.

4. ACKNOWLEDGEMENTS

Ged borrows freely from the ideas and code of the *gex* program by D. J. Jackowski. The first version of *ged* was written by D. E. Pinkston.

5. REFERENCES

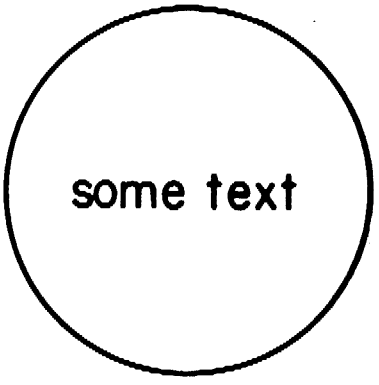
[1] Feuer, A. R. *UNIX Graphics Overview*, Bell Laboratories (1979).

[2] Dolotta, T. A., Olsson, S. B., and Petruccelli, A. G. (eds.). *UNIX User's Manual*—Release 3.0, Bell Laboratories (June 1980).

APPENDIX: Some Examples of What Can be Done

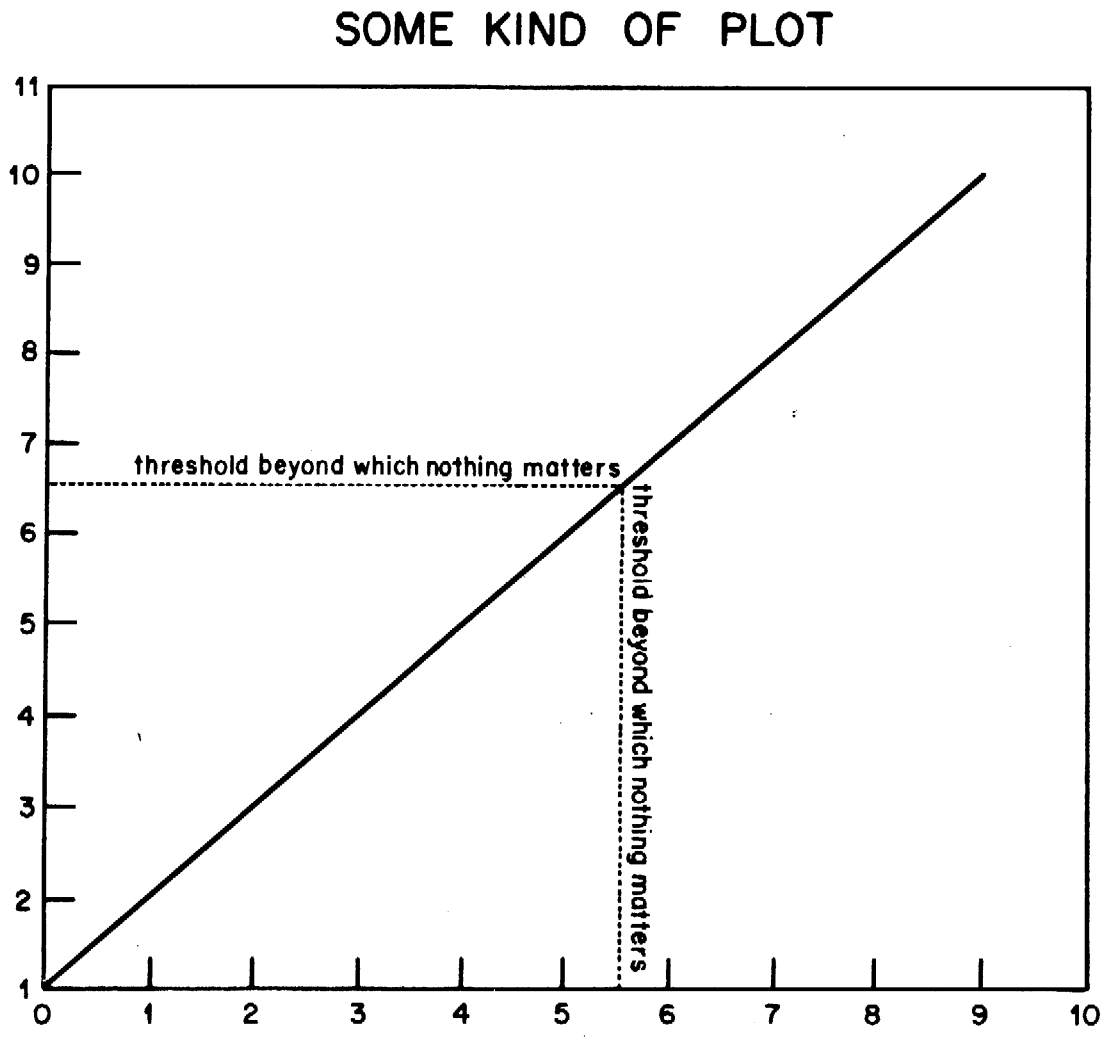
1. Text Centered Within a Circle

*Circle <cr>	
<position cursor> <sp>	(establish center)
<position cursor> <cr>	(establish radius)
*Text −m <cr>	(text is to be centered)
some text <cr>	
$.0 <cr>	(first point from previous set, i.e., circle center)
<cr>	



2. Making Notes on a Plot

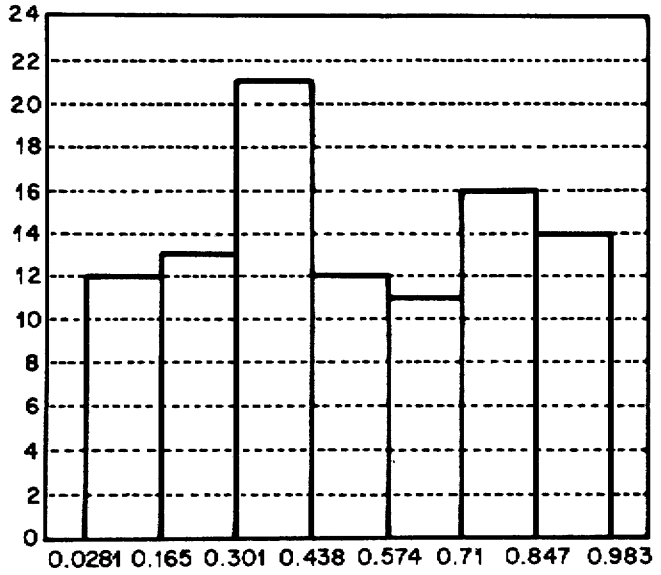
***! gas | plot −g >A <cr>**	(generate a plot, put it in file A)
***read −e− A <cr>**	(input the plot, but do not display it)
***view −h <cr>**	(window on the plot)
***Lines −sdo <cr>**	(draw dotted lines)
<position cursor> <sp>	
<position cursor> <sp>	
<position cursor> <sp>	
<cr>	(end of Lines)
***set −h150,wn <cr>**	(set text height to 150, line width to narrow)
***Text −r <cr>**	(right justify text)
threshold beyond which nothing matters <cr>	
<position cursor> <cr>	(set right point of text)
***Text −a−90 <cr>**	(rotate text negative 90 degrees)
threshold beyond which nothing matters <cr>	
<position cursor> <cr>	(set top end of text)
***x <cr>**	(find center of plot)
<position cursor> <sp>	(top left of plot)
<position cursor> <cr>	(bottom right)
***Text −h300,wm,m <cr>**	(build title: height 300, weight medium, centered)
SOME KIND OF PLOT <cr>	
<position cursor> <cr>	(set title centered above plot)



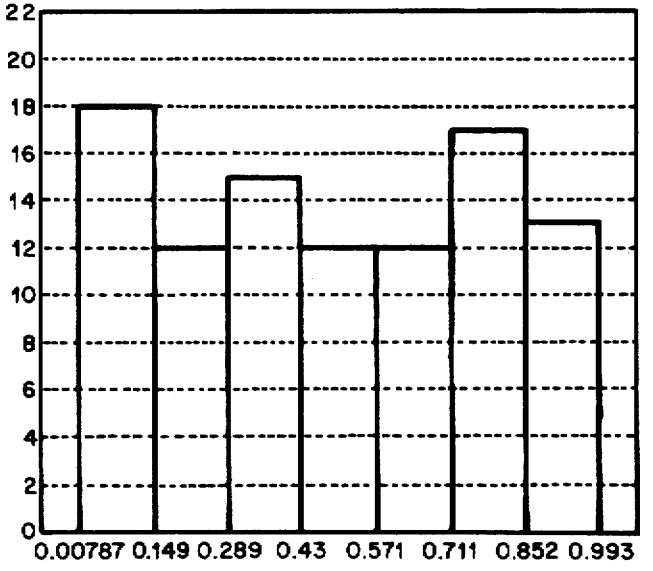
3. A Page Layout with Drawings and Text

```
*! rand −s1,n100 | title −v"seed 1" | qsort | bucket | hist −r12 >A <cr>
        (put a histogram, region 12, of 100 random numbers into file A)
*! rand −s2,n100 | title −v"seed 2" | qsort | bucket | hist −r13 >B <cr>
        (put another histogram, region 13, into file B)
*! ed <cr>                          (create a file of text using the text editor)
a <cr>
On this page are two histograms <cr>
from a series of 40 <cr>
designed to illustrate the weakness <cr>
of multiplicative congruential random number generators.  <cr>
.pl \n(nlu <cr>                     (mark end of page)
.  <cr>
w C <cr>                            (put the text into file C)
156
q <cr>
*! nroff C | yoo C <cr>             (format C, leave the output in C)

*view −u <cr>                       (window on the universe)
*read A <cr>
*read B <cr>
*read −h300,wn,m C <cr>             (text height 300, line weight narrow, text centered)
<position cursor> <cr>              (center text over two plots)
*view −h <cr>                       (window on the resultant drawing)
```

On this page are two histograms from a series of 40 designed to illustrate the weakness of multiplicative congruential random number generators.



SEED 1



SEED 2

STAT—A Tool for Analyzing Data

A. R. Feuer
A. Guyton

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Stat is a collection of numerical programs under the UNIX† operating system that can be interconnected using the shell [1] to form processing networks. Included within *stat* are programs to generate simple statistics and pictorial output.

This paper introduces *stat* concepts and commands through a collection of examples. A complete definition of each command is given.

1. INTRODUCTION

Much of the power for manipulating text under UNIX comes from the numerous well defined text processing programs that can be readily interfaced to one another. The general interface is an unformatted text string and the interconnection mechanism is usually the shell [1]. Because the programs are independent from one another, new functions can easily be added and old ones changed. And because the text editor also operates on unformatted text, arbitrary text manipulation can always be performed even when the more specialized routines are insufficient.

Stat uses the same mechanisms to bring a similar power to the manipulation of numbers. It consists of a collection of numerical processing routines that read and write unformatted text strings. It includes programs to build graphical files that can be manipulated using a graphical editor. And since *stat* programs process unformatted text, they can readily be connected with other UNIX command-level (i.e., callable from shell) routines.

It is useful to think of the shell as a tool for constructing processing networks in the sense of data flow programming. Command-level routines are the nodes of the network and pipes and tees are the links. Data flows from node to node in the network via links.

Section 2 of this paper is a introduction to the concepts of *stat*. Section 3 contains a description of each of the nodes. A few examples of *stat* usage are given in an appendix.

2. BASIC CONCEPTS

All numerical data in *stat* is of type *vector*. A *vector* is a sequence of numbers separated by delimiters. Vectors are processed by command-level routines called *nodes*.

2.1 Transformers

A *transformer* is a node that reads an input element, operates upon it, and outputs the resulting value. For example, suppose file **A** contains the vector

```
1 2 3 4 5
```

then the command

```
root A (typed input is bold)
```

produces

---

† UNIX is a trademark of Bell Laboratories.

```
1    1.41421    1.73205    2    2.23607
```

the square root of each input element. Analogously,

```
log A
```

produces

```
0    0.693147    1.09861    1.38629    1.60944
```

the natural logarithm of each element of vector *A*.

Af, for arithmetic function, is a particularly versatile transformer. Its argument is an expression that is evaluated once for each complete set of input values. A simple example is

```
af "2*A^2"
```

which produces

```
2    8    18    32    50
```

twice the square of each element from *A*. Expression arguments to **af** are usually surrounded by quotes since some of the operator symbols have special meaning to the shell.

2.2 Summarizers

A *summarizer* is a node that calculates a statistic for a vector. Typically, summarizers read in all of the input values, then calculate and output the statistic. For example, using the vector *A* from above,

```
mean A
```

produces

```
3
```

and

```
total A
```

produces

```
15
```

2.3 Parameters

Most nodes accept parameters to direct their operation. Parameters are specified as command-line options. **Root**, for example, is more general than just square root, any root may be specified using the **r** option. Thus

```
root -r3 A
```

produces

```
1    1.25992    1.44225    1.5874    1.70998
```

the cube root of each element from *A*.

2.4 Building Networks

Nodes are interconnected using standard shell concepts and syntax. Pipes are the linear connector attaching the output of one node to the input of another. As an example, to find the mean of the cube roots of vector *A* is simply

```
root -r3 A | mean
1.39991
```

Often the required network is not so simple. Tees and sequence can be used to build nonlinear networks. To find the mean and median of the transformed vector *A* is

```
root −r3 A | tee B | mean; point B
1.399
1.442
```

Beware of the distinction between the sequence operator, ";", and the linear connector, the pipe. Because processes in a pipeline run concurrently, each file name in the pipeline must be unique. Sequence implies run to completion (so long as "&" isn't used) hence names may be duplicated, and often are.

There is a special case of nonlinear networks where the result of one node is used as command-line input for another. Command substitution makes this easy. For example, to generate residuals from the mean of *A* is simply

```
af "A−`mean A`"
−2    −1    0     1     2
```

2.5 Vectors, a Closer Look

Thus far we have used vectors, but not created them. One way to create a vector is by using a *generator*. A *generator* is a node that accepts no input and outputs a vector based upon definable parameters. *Gas* is a generator that produces additive sequences. One of the parameters to *gas* is the number of elements in the generated vector. As an example, to create the vector *A* that we have been using is

```
gas −n5
1     2     3     4     5
```

Vectors are, however, merely text files. Hence we could use the text editor to create and modify the same vector.

A useful property of vectors is that they consist of a sequence of numbers surrounded by delimiters, where a delimiter is anything that is not a number. (Numbers are constructed in the usual way: [sign](digits)(.digits)[e[sign]digits], where fields are surrounded by brackets and parentheses. All fields are optional, but at least one of the fields surrounded by parentheses must be present.) Thus vector *A* could also be created by building the file **B** in the text editor as

```
1partridge,2tdoves,3frhens,4cbirds,5gldnrings,
```

which when read yields

```
list B
1     2     3     4     5
```

A note should be made as to the size of a vector: vectors are as long as they are. That is, a vector is a stream containing numbers terminated by an EOF (EOT from the keyboard). A good illustration of this is to use the keyboard as the source of the input vector, as in

```
cusum -c1
2   <return>
2
16.3  <return>
18.3
25.4  <return>
43.7
14  <return>
57.7
<cntrl d>
```

which implements a running accumulator. Since no vector was given to **cusum**, the input is taken from the standard input until an EOT.

2.6 A Simple Example: Interacting with a Data Base

When used in conjunction with UNIX tools for manipulating text *stat* provides an effective means for exploring a numerical data base. Suppose, for example, we have a subdirectory called **data** containing data files that include the lines:

```
path length = nn      (nn is any number)
node count = nn
```

Then we can access the value for **node count** from each file, sort the values into ascending order, store the resulting vector in file **A**, and get a copy on the terminal by typing

```
grep "node count" data/* | qsort | tee A
17    19    22    32    39
50    68    78    125   139
```

Note that if some of the data files have numbers in their name, we must protect against those numbers from being considered data. Using **cat** this is easy:

```
cat data/* | grep "node count" | qsort | tee A
```

To get a feel for the distribution of node counts shell iteration can be used to advantage.

```
for i in .25 .5 .75
do point -pSi A
done
24.5
44.5
75.5
```

generates the lower hinge, the median, and the upper hinge of the sorted vector *A*.

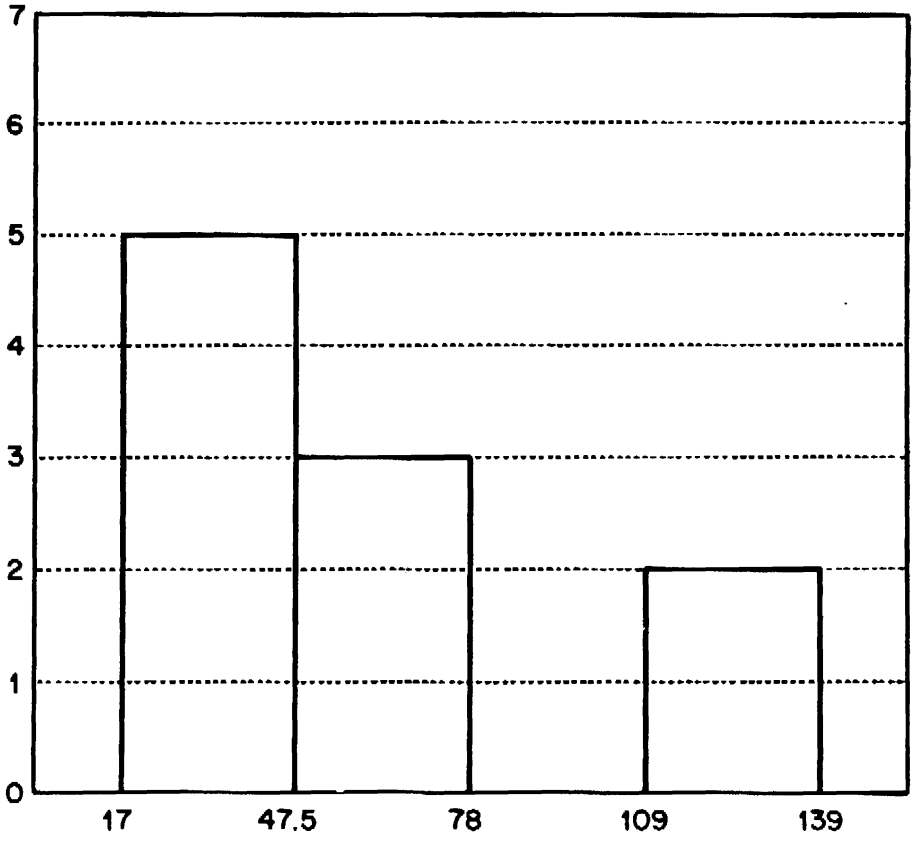
2.7 Translators

Translators are used to view data pictorially. A *translator* is a node that produce a stream of a different structure from that which it consumes. Graphical translators consume vectors and produce pictures in a language called GPS, for graphical primitive string. (Among the programs that understand GPS is *ged*, the graphical editor [2], which means that the graphical output of any translator can be directly edited at a display terminal.) **Hist** is an example of a translator; it produces a GPS that describes a histogram from input consisting of interval limits and counts. The summarizer **bucket** produces limits and counts, thus

```
bucket A | hist | td
```

generates a histogram of the data of vector *A* and displays it on a display terminal (Fig. 1). **Td** translates the GPS into machine code for Tektronix 4010 series display terminals.

Figure 1. bucket A | hist | td



A wide range of X-Y plots can be constructed using the translator **plot**. For example, to build a scatter plot of **path length** with **node count** (Fig. 2) is

```
grep "path length" data/* | title -v"path length" >A
grep "node count" data/* | title -v"node count" | plot -FA,dg | td
```

A vector may be given a title using **title**. When a titled vector is plotted the appropriate axis is labeled with the vector title. When a titled vector is passed through a transformer the title is altered to reflect the transformation. Thus in a graph of log **node count** versus the cube root of **path length**, i.e.,

```
grep "node count" | title -v"node count" | log >B
root -r3 A | plot -F-,dg B | td
```

the axis labels automatically agree with the vectors plotted (Fig. 3).

3. NODE DESCRIPTIONS

The *stat* nodes are divided into four classes: *transformers*, *summarizers*, *translators*, and *generators*. In this section a description of each node is given. The descriptions are organized by node class.

Figure 2. Scatter plot

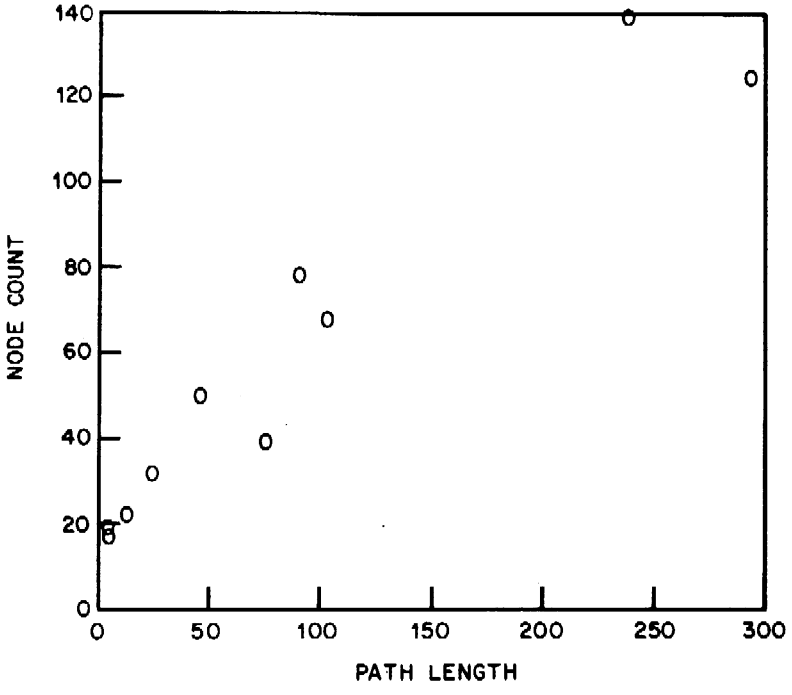
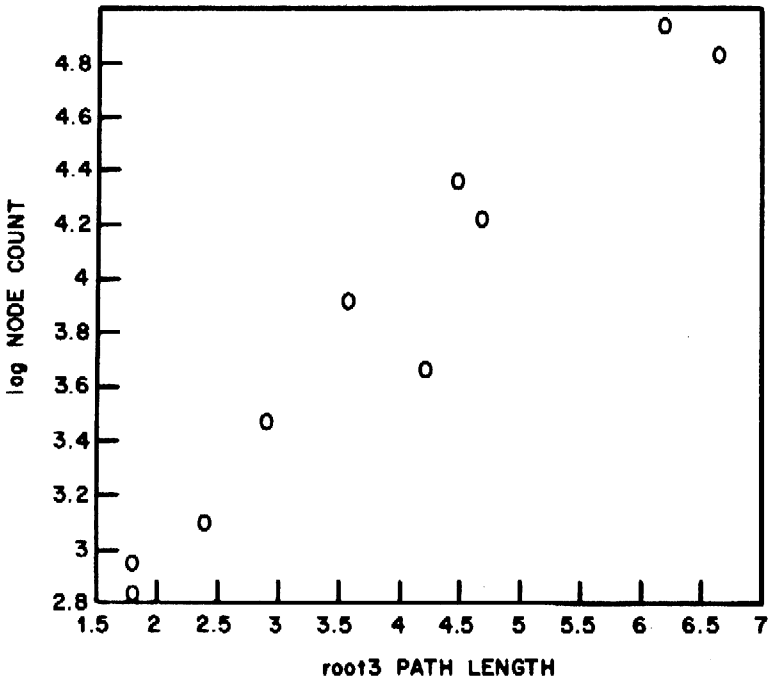


Figure 3. Transformed scatter plot



All of the nodes accept the same command-line format:

A *command* is	a *command-name* followed by zero or more *arguments.*
A *command-name* is	the name of any *stat* node.
An *argument* is	a *file-name* or an *option-string.*
An *option-string* is	a − followed by one or more *options.*
An *option* is	one or more letters followed by an optional value. Options may be separated by commas.
A *file-name* is	any name not beginning with −, or a − by itself (to reference the standard input).

Each file argument to a node is taken as input to one occurrence of the node. That is, the node is executed from its initial state once per file. If no files are given, the standard input is used. All nodes, except generators, accept files as input, hence it is not made explicit in the synopses that follow.

Most nodes accept command-line *options* to direct the execution of the node. Some *options* take values. In the following synopses, to indicate the type of value associated with an *option*, the *option* key-letter is followed by:

i	to indicate integer,
f	to indicate floating point or integer,
string	to indicate a character string, or
file	to indicate a *file-name*

Thus the *option* **c***i*, implies **c** expects an integer value (**c** := *integer*).

3.1 Transformers

Transformers have the form

V_{in} *transform* V_{out}

where, by convention, V_{in} is a vector Y, with elements y_1 through y_k ($y_{1:k}$) and V_{out} is a vector Z, $z_{1:m}$. All transformers have a **c***i* option, where **c** specifies the number of columns per line in the output. By default, **c** := 5.

abs − absolute value

$z_i := |y_i|$

af [−**t** **v**] − arithmetic function

The command-line format of **af** is an extension of the command-line description given above, with *expression* replacing *file-name*; an *expression* consists of *operands* and *operators.*

An *operand* is either a *vector, function, constant,* or *expression*:

A *vector* is	a file name with the restriction that file names begin with a letter, and are composed only of letters, digits, ".", and "_". The first unknown file name (one not in the current directory) references the standard input.
A *function* is	the name of a command followed by its arguments in parentheses. Arguments are written in command-line format.
A *constant* is	an integer or floating point (but not "E" notation) number.

The *operators* are listed below in order of decreasing precedence. Parentheses may be used to alter precedence. x_i (y_i) represents the start element from X (Y) for the expression.

$'Y$	reference y_{i+1}. y_{i+1} is consumed; the next value from Y is y_{i+2}. Y is a vector.
$X \hat{} Y \quad -Y$	x_i raised to the y_i power, negation of y_i. Association is right to left. X and Y are expressions.
$X*Y \quad X/Y \quad X\%Y$	x_i multiplied by, divided by, modulo y_i. Association is left to right. X and Y are expressions.
$X+Y \quad X-Y$	x_i plus, minus y_i. Association is left to right. X and Y are expressions.
X,Y	yields x_i, y_i. Association is left to right. X and Y are expressions.

Options:

t	causes the output to be titled from the vector on the standard input.
v	causes function expansions to be echoed.

ceil — ceiling

$z_i :=$ smallest integer greater than y_i

cusum — cumulative sum

$$z_i := \sum_{j=1}^{i} y_j$$

exp — exponential function

$z_i := e^{y_i}$

floor — floor

$z_i :=$ largest integer less than y_i

gamma — gamma function

$z_i := \Gamma(y_i)$

list [−**d***string*] — list vector elements

$z_i := y_i$

If **d** is not specified, then any character that is not part of a number is a delimiter. If **d** is specified, then the white space characters (space, tab, and new-line) plus the character(s) of *string* are delimiters. Only numbers surrounded by delimiters are listed.

log [−**b***f*] — logarithmic function

$z_i := \log_{\mathbf{b}} y_i$

By default, **b** $:= e$ $\quad (e \approx 2.71828\ldots)$

mod [−**m***f*] − modulus

$z_i := y_i$ modulo **m**

By default, **m** := 2

pair [−**F***file* **x***i*] − pair elements

F is a vector X, $x_{1:j}$, and **x** is the number of elements per group from X. Let % denote modulo and / denote integer division, then

$$z_i := \begin{cases} y_{(i/(\mathbf{x}+1))} & \text{if } i\%(\mathbf{x}+1) = 0 \\ x_{(i-i/(\mathbf{x}+1))} & \text{if } i\%(\mathbf{x}+1) \neq 0 \end{cases}$$

$rank(Z) = (\mathbf{x}+1)minimum(k, j/\mathbf{x})$

If **F** is not specified, then X comes from the standard input.
If both X and Y come from the standard input, X precedes Y.

By default, **x** := 1

power [−**p***f*] − raise to a power

$z_i := y_i^{\mathbf{p}}$

By default, **p** := 2

root [−**r***f*] − extract a root

$z_i := \sqrt[\mathbf{r}]{y_i}$

By default, **r** := 2

round [−**p***i* **s***i*] − round off values

if **s** is specified, then

$z_i := y_i$ rounded up to **s** significant digits,

else if **p** is specified, then

$z_i := y_i$ rounded up to **p** digits beyond the decimal point.

By default, **p** := 0

siline [−**i***f* **n***i* **s***f*] − generate a line, given a slope and intercept

$z_i = \mathbf{s}\, y_i + \mathbf{i}$

if **n** is specified, then

$Y = 0, 1, 2, 3, \cdots, \mathbf{n}$.

By default, **i** := 0, **s** := 1

sin − sine function

$z_i := \sin(y_i)$

spline [−options] − interpolate smooth curve

Y and Z are sequences of X,Y coordinates (like that produced by **pair**).

For more information about **spline**, see *spline*(1) in the *UNIX User's Manual* [4].

subset [−**a***f* **b***f* **F***file* **i***i* **l***f* **n**l **np p***f* **s***i* **t***i*] − generate a subset

Z consists of elements selected from Y. Selection occurs as follows:

Let C(w) be true if

($w >$ **a** or $w <$ **b** or $w =$ **p**) and $w \neq$ **l**

is true. If neither **a**, **b**, nor **p** are specified, C(w) is true if $w \neq$ **l** is true.

CASE 1 − **nl** or **np** not specified.

If **F** is specified, then $key_i = x_i$
else $key_i = y_i$.

For $r =$ **s**, **s**+**i**, **s**+2**i**, $\cdots$ with $r \leq$ **t**,
y_r becomes an element of Z if C(key_r) is true.

By default, **i** := 1, **s** := 1, **t** := 32767.

CASE 2 − **np** is specified.

F is a vector X, $x_{1:j}$.
For $r = x_1, x_2, \cdots, x_j$,
y_r becomes an element of Z if C(y_r) is true.

CASE 3 − **nl** is specified.

F is a vector X, $x_{1:j}$.
For $r \neq x_1, x_2, \cdots, x_j$,
y_r becomes an element of Z if C(y_r) is true.

For cases 2 and 3, if **F** is not specified then the standard input is used for X. Either X or Y may come from the standard input, but not both.

3.2 Summarizers

Summarizers have the form

V_{in} *summarize* V_{out}

where, again, V_{in} is a vector Y, $y_{1:k}$, and V_{out} is a vector Z, $z_{1:m}$. For many summarizers, $rank(Z) = 1$.

bucket [−**a***i* **c***i* **F***file* **h***f* **i***i* **l***f* **n***i*] − break into buckets

Y must be a sorted vector.
Z consists of odd elements (parenthesized) which are bucket limits and even elements which are bucket counts.

The count is the number of elements from Y greater than the lower limit (greater than or equal to for the lowest limit), and less than or equal to the higher limit. If specified, the limit values are taken from **F**. Otherwise the limits are evenly spaced between **l** and **h** with a total of **n** buckets. If **n** is not specified, the number of buckets is determined as follows:

$$\mathbf{n} := \begin{cases} \dfrac{\mathbf{h} - \mathbf{l}}{\mathbf{i}} & \text{if } \mathbf{i} \text{ is specified} \\ \dfrac{k}{\mathbf{a} + 1} & \text{if } \mathbf{a} \text{ is specified} \\ 1 + \log_2 k & \text{if neither } \mathbf{a} \text{ nor } \mathbf{i} \text{ are specified.} \end{cases}$$

c specifies the number of columns in the output.

By default:

c := 5
h := largest element of Y
l := smallest element of Y

cor [−**F***file*] − correlation coefficient

If **F** is a vector X, $x_{1:k}$, let $\bar{x} = \dfrac{\sum_{i=1}^{k} x_i}{k}$ and $\bar{y} = \dfrac{\sum_{i=1}^{k} y_i}{k}$, then

$$z_1 := \frac{\sum_{i=1}^{k} (x_i - \bar{x})(y_i - \bar{y})}{\left[\sum_{i=1}^{k} (x_i - \bar{x})^2\right]^{1/2} \left[\sum_{i=1}^{k} (y_i - \bar{y})^2\right]^{1/2}}$$

X and Y must have the same rank. If **F** is not specified, the standard input is used for X. If both X and Y come from the standard input, X precedes Y.

hilo [−**h l o ox oy**] − high and low values

z_1 := lowest value across all input vectors

z_2 := highest value across all input vectors

Options to control output:

h	Only output high value.
l	Only output low value.
o	Output high, low values in option form (suitable for **plot**).
ox	Output high, low values with "x" prefixed.
oy	Output high, low values with "y" prefixed.

lreg [−**F***file* **i o s**] − linear regression

If **F** is a vector X, $x_{1:k}$, let $\bar{x} = \dfrac{\sum_{i=1}^{k} x_i}{k}$ and $\bar{y} = \dfrac{\sum_{i=1}^{k} y_i}{k}$, then

$$z_1 := \bar{y} - z_2\bar{x} \qquad \text{(intercept)}$$

and

$$z_2 := \frac{\dfrac{\sum_{i=1}^{k} x_i y_i}{k} - \bar{x}\bar{y}}{\dfrac{\sum_{i=1}^{k} x_i^2}{k} - \bar{x}^2} \qquad \text{(slope)}$$

X and Y must have the same rank.
If **F** is not specified, then

$$X = 0, 1, 2, \cdots, k$$

Options to control output:

i	Only output the intercept.
o	Output the slope and intercept in option form (suitable for **siline**).
s	Only output the slope.

mean [−**f***f* **n***i* **p***f*] − (trimmed) mean

$$z_1 := \frac{\sum_{i=1}^{k} y_i}{k}$$

Y may be trimmed by

(1/**f**) k	elements from each end,
p *k*	elements from each end, or
n	elements from each end.

By default, **n** := 0

point [−**f***f* **n***i* **p***f* **s**] − empirical cumulative density function point

z_1 := linearly interpolated Y value corresponding to the

100 (1/**f**)	percent point, the
100 **p**	percent point, or the
nth	element.

Negative option values are taken from the high end of Y. Option **s** implies Y is sorted.

By default, **p** := .5 (median)

prod − product

$$z_1 := \prod_{i=1}^{k} y_i$$

qsort [−**c***i*] − quicksort

z_i := ith smallest element of Y.

By default, **c** := 5

rank − rank

z_1 := number of elements in Y.

total − sum

$$z_1 := \sum_{i=1}^{k} y_i$$

var − variance

$$z_1 := \frac{\sum_{i=1}^{k} (y_i - \bar{y})^2}{k - 1}$$

3.3 Translators

Translators have the form

F_{in} *translate* F_{out}

where F_{in} may be a vector or a GPS depending upon the translator. F_{out} is a GPS. A *GPS* (Graphical Primitive String) is a format for storing a picture. A picture is defined in a Cartesian plane of 64K points on each axis. The plane, or universe, is divided into 25 square regions

numbered 1 to 25 from the lower left to the upper right. Various commands exist that can display and edit a GPS. For more information, see *graphics*(1) in the *UNIX User's Manual* [4] and *UNIX Graphics Overview* [3].

bar [**−a b f g r***i* **w***i* **x***f* **xa y***f* **ya yl***f* **yh***f*] − build a bar chart

F_{in} is a vector, each element of which defines the height of a bar. By default, the x-axis will be labeled with positive integers beginning at 1; for other labels, see **label**.

Options:

a	Suppress axes.
b	Plot bar chart with bold weight lines, otherwise use medium.
f	Do not build a frame around plot area.
g	Suppress background grid.
r*i*	Put the bar chart in GPS region *i*, where *i* is between 1 and 25 inclusive. The default is 13.
w*i*	*i* is the ratio of the bar width to center-to-center spacing expressed as a percentage. Default is 50, giving equal bar width and bar space.
x*f* (**y***f*)	Position the bar chart in the GPS universe with x-origin (y-origin) at *f*.
xa (**ya**)	Do not label x-axis (y-axis).
yl*f*	*f* is the y-axis low tick value.
yh*f*	*f* is the y-axis high tick value.

hist [**−a b f g r***i* **x***f* **xa y***f* **ya yl***f* **yh***f*] − build a histogram

F_{in} is a vector (of the type produced by **bucket**) of odd rank, with odd elements being limits and even elements being bucket counts.

Options:

a	Suppress axes.
b	Plot histogram with bold weight lines, otherwise use medium.
f	Do not build a frame around plot area.
g	Suppress background grid.
r*i*	Put the histogram in GPS region *i*, where *i* is between 1 and 25 inclusive. The default is 13.
x*f* (**y***f*)	Position the histogram in the GPS universe with x-origin (y-origin) at *f*.
xa (**ya**)	Do not label x-axis (y-axis).
yl*f*	*f* is the y-axis low tick value.
yh*f*	*f* is the y-axis high tick value.

label [**−b c F***file* **h p r***i* **x xu y yr**] − label the axis of a GPS file

F_{in} is a GPS of a data plot (like that produced by **hist, bar,** and **plot**). Each line of the *label file* is taken as one label. Blank lines yield null labels. Either the GPS or the *label file*, but not both, may come from the standard input.

Options:

b	Assume the input is a bar chart.
c	Retain lower case letters in labels, otherwise all letters are upper case.
F*file*	*file* is the *label file*.
h	Assume the input is a histogram.
p	Assume the input is an x-y plot. This is the default.
r*i*	Labels are rotated *i* degrees. The pivot point is the first character.
x	Label the x-axis. This is the default.
xu	Label the upper x-axis, i.e., the top of the plot.
y	Label the y-axis.
yr	Label the right y-axis, i.e., the right side of the plot.

pie [**−b o p pn***i* **pp***i* **r***i* **v x***i* **y***i*] − build a pie chart

F_{in} is a vector with a restricted format. Each input line represents a slice of pie and is of the form:

[< **i e f c***color* >] value [label]

with brackets indicating optional fields. The control field options have the following effect:

i	The slice will not be drawn, though a space will be left for it.
e	The slice is "exploded," or moved away from the pie.
f	The slice is filled. The angle of fill lines depends on the color of the slice.
c*color*	The slice is drawn in *color* rather than the default black. Legal values for *color* are **b** for black, **r** for red, **g** for green, and **u** for blue.

The pie is drawn with the *value* of each slice printed inside and the *label* printed outside.

Options:

b	Draw pie chart in bold weight lines, otherwise use medium.
o	Output values around the outside of the pie.
p	Output *value* as a percentage of the total pie.
pn*i*	Output *value* as a percentage, but total of percentages equals *i* rather than 100. **pn**100 is equivalent to **p**.
pp*i*	Only draw *i* percent of a pie.
r*i*	Put the pie chart in region *i*, where *i* is between 1 and 25 inclusive. The default is 13.
v	Do not output values.
x*i* (**y***i*)	Position the pie chart in the GPS universe with x-origin (y-origin) at *i*.

plot [**−a b c***string* **d f F***file* **g m r***i* **x***f* **xa xh***f* **xi***f* **xl***f* **xn***i* **xt y***f* **ya yh***f* **yi***f* **yl***f* **yn***i* **yt**] − plot a graph

F_{in} is a vector(s) which contains the y values of an x-y graph. Values for the x-axis come from F. Axis scales are determined from the first vector plotted.

Options:

a	Suppress axes.
b	Plot graph with bold weight lines, otherwise use medium.
c*string*	The character(s) of *string* are used to mark points. Characters from *string* are used, in order, for each separately plotted graph included in the plot. If the number of characters in *string* is less than the number of plots, the last character will be used for all remaining plots. The **m** option is implied.
d	Do not connect plotted points, implies option **m**.
f	Do not build a frame around plot area.
F*file*	Use *file* for x-values, otherwise the positive integers are used. This *option* may be used more than once, causing a different set of x-values to be paired with each input vector. If there are more input vectors than sets of x-values, the last set applies to the remaining vectors.
g	Suppress the background grid.
m	Mark the plotted points.
r*i*	Put the graph in GPS region *i*, where *i* is between 1 and 25 inclusive. The default is 13.
x*f* (**y***f*)	Position the graph in the GPS universe with x-origin (y-origin) at *f*.
xa (**ya**)	Omit x-axis (y-axis) labels.
xh*f* (**yh***f*)	*f* is the x-axis (y-axis) high tick value.
xi*f* (**yi***f*)	*f* is the x-axis (y-axis) tick increment.
xl*f* (**yl***f*)	*f* is the x-axis (y-axis) low tick value.
xn*i* (**yn***i*)	*i* is the approximate number of ticks on the x-axis (y-axis).

xt (yt) Omit x-axis (y-axis) title.

title [−**b c l***string* **v***string* **u***string*] − title a vector or GPS

F_{in} can be either a GPS or a vector with F_{out} being of the same type as F_{in}. **Title** prefixes a *title* to a vector or appends a *title* to a GPS.

Options apply as indicated:

b	Make the GPS *title* bold.
c	Retain lower case letters in *title*, otherwise all letters are upper case.
l*string*	For a GPS, generate a lower *title* := *string*.
u*string*	For a GPS, generate an upper *title* := *string*.
v*string*	For a vector, *title* := *string*.

3.4 Generators

Generators have the form

generate V_{out}

where V_{out} is a vector Z, $z_{1:k}$. All generators have a **c***i* option where **c** specifies the number of columns per line in the output. By default, **c** := 5.

gas [−**i***f* **n***i* **s***f* **t***f*] − generate additive sequence

Z is constructed as follows:

$$z_1 := \mathbf{s}$$

$$z_{i+1} := \begin{cases} z_i + \mathbf{i} & \text{if } |z_i| \leq \mathbf{t} \\ z_1 & \text{otherwise} \end{cases}$$

$rank(Z) = \mathbf{n}$.

By default, **i** := 1, **n** := 10, **s** := 1, **t** := ∞

prime [−**h***i* **l***i* **n***i*] − generate prime numbers

The elements of Z are consecutive prime numbers with

$$\mathbf{l} \leq z_i \leq \mathbf{h}$$

$rank(Z) \leq \mathbf{n}$.

By default, **n** := 10, **l** := 2, **h** := ∞

rand [−**h***f* **l***f* **m***f* **n***i* **s***i*] − generate random sequence

The elements of Z are random numbers generated by a multiplicative congruential generator with **s** acting as a seed, such that

$$\mathbf{l} \leq z_i \leq \mathbf{h}$$

If **m** is specified, then

$$\mathbf{h} = \mathbf{m} + \mathbf{l}$$

$rank(Z) = \mathbf{n}$.

By default, **h** := 1, **l** := 0, **n** := 10, **s** := 1

REFERENCES

[1] S. R. Bourne. *An Introduction to the UNIX Shell,* Bell Laboratories.

[2] A. R. Feuer. *A Tutorial Introduction to the Graphical Editor,* Bell Laboratories.

[3] A. R. Feuer. *UNIX Graphics Overview,* Bell Laboratories.

[4] T. A. Dolotta, S. B. Olsson, and A. G. Petruccelli (eds.). *UNIX User's Manual*—Release 3.0, Bell Laboratories (June 1980).

APPENDIX

- Example 1:

PROBLEM

Calculate the total value of an investment held for a number of years at an interest rate compounded annually.

SOLUTION

```
Principal=1000
echo Total return on $Principal units compounded annually
echo "rates:\t\t\c"; gas -s.05,t.15,i.03 | tee rate
for Years in 1 3 5 8
do
        echo "$Years year(s):\t\c"; af "$Principal*(1+rate)^$Years"
done
```

Total return on 1000 units compounded annually

rates:	0.05	0.08	0.11	0.14
1 year(s):	1050	1080	1110	1140
3 year(s):	1157.62	1259.71	1367.63	1481.54
5 year(s):	1276.28	1469.33	1685.06	1925.41
8 year(s):	1477.46	1850.93	2304.54	2852.59

NOTES

Notice the distinction between vectors and constants as operands in the expression to **af**. The shell variables **$Principal** and **$Years** are constants to **af**, while the file **rate** is a vector. **Af** executes the expression once per element in **rate**.

- Example 2:

PROBLEM

Given are three ordered vectors (*A*, *B*, and *C*) of scores from a number of tests. Each vector is from one test-taker, each element in a vector is the score on one test. There are missing scores in each vector indicated by the value −1. Generate three new vectors containing scores only for those tests where no data is missing.

SOLUTION

```
echo Before:
gas -n^rank A^ | tee N | af "label,A,B,C"

for i in N B C A
do subset -FA,l-1 $i >s$i; done
for i in N A C B
do subset -FsB,l-1 s$i | yoo s$i; done
for i in N A B C
do subset -FsC,l-1 s$i | yoo s$i; done

echo "\nAfter:"
af "sN,sA,sB,sC"
```

Before:

1	5	6	−1
2	7	10	10
3	−1	10	9
4	10	−1	8
5	6	5	−1
6	5	7	5
7	−1	7	8
8	−1	−1	8
9	3	−1	8
10	6	10	10
11	7	5	7

After:

2	7	10	10
6	5	7	5
10	6	10	10
11	7	5	7

NOTES

The approach is to eliminate those elements in all vectors that correspond to −1 in the base vector. Each of the three vectors takes turn at being the base. It is important that the base be subsetted last. The command **yoo** (see *gutil*(1) [4]) takes the output of a pipeline and copies it into one of the files used in the pipeline. This cannot be done by redirecting the output of the pipeline as this would cause a concurrent read and write on the same file.

The printing of the "Before" matrix illustrates a useful property of **af**. The first name in an expression that does not match any name in the present working directory is a reference to the standard input. In this example, **label** references the input coming through the pipe.

- Example 3:

PROBLEM

Generate a bar chart of the percent of execution time consumed by each routine in a program.

SOLUTION

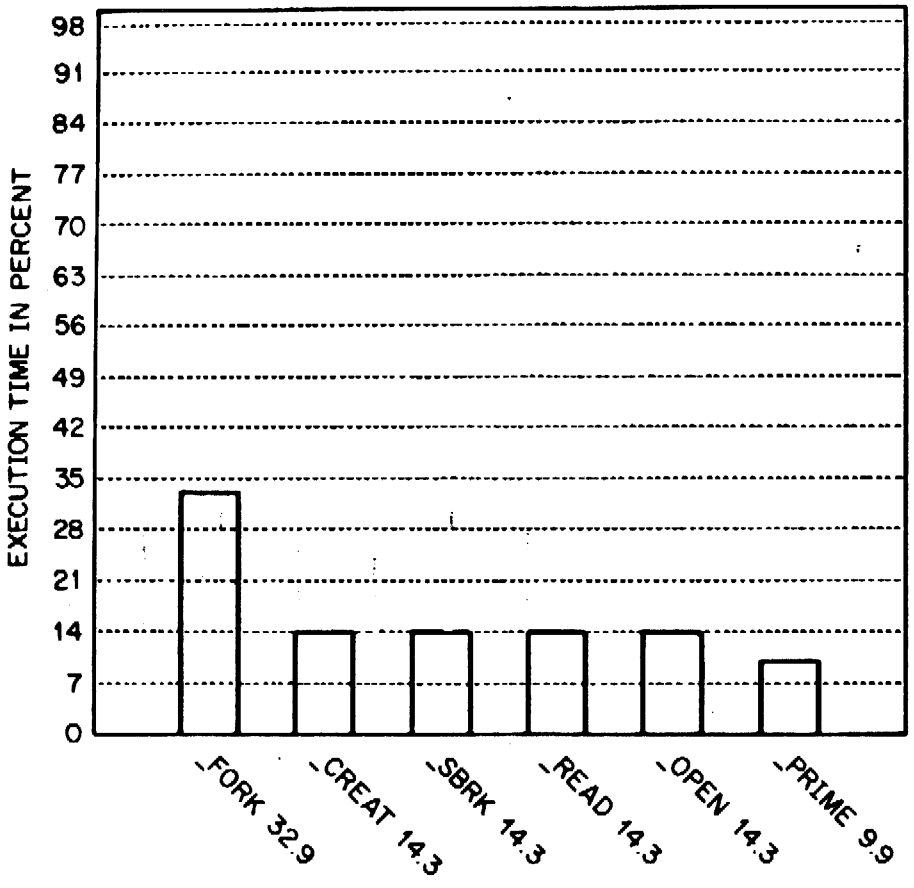
```
prof | cut -c1 -15 | sed -e 1d -e "/ 0.0/d" -e "s/^  *//" >P
echo These are the execution percentages; cat P
title P -v"execution time in percent" | bar -xa -yl0,yh100 |
        label -br-45,FP | td
```

These are the execution percentages
_fork 32.9
_creat 14.3
_sbrk 14.3
_read 14.3
_open 14.3
_prime 9.9

NOTES

Prof is a UNIX command that generates a listing of execution times. **Cut** and **sed** are used to eliminate extraneous text from the output of **prof**. (It is because verbiage can get in the way that *stat* nodes say very little.) Notice that **P** is a vector to **title** while it is a text file to **cat** and **label**.

Figure a1



- Example 4:

PROBLEM

Plot the relationship between the execution time of a program and the number of processes in the process table.

SOLUTION

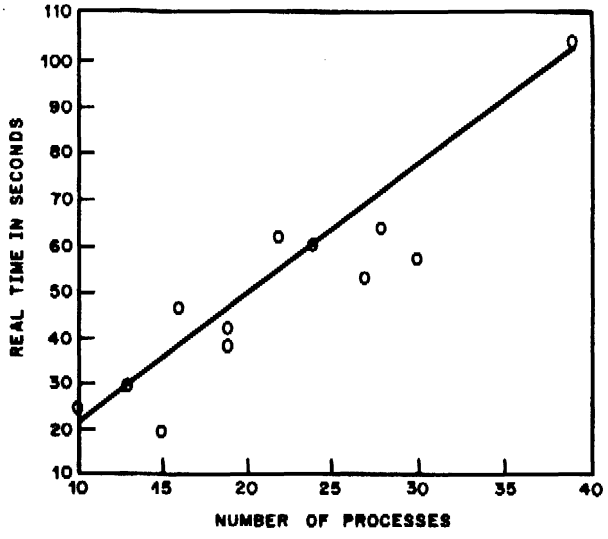
```
# The first program generates the performance data
for i in `gas -n12`
do
        ps -ae | wc -l >>Procs&
        time prime -n1000 >/dev/null 2>>Times
        sleep 300
done

# The second program analyzes and plots the data
for i in real user sys
do
        grep $i Times | sed "s/$i//" |
                awk -F: "{ if(NF==2) print \$1*60+\$2; else print }" |
                        title -v"$i time in seconds" >$i
        siline -`lreg -o,FProcs $i` Procs >$i.fit
done
title -v"number of processes" Procs | yoo Procs

plot -dg,FProcs real -r12 >R12
plot -ag,FProcs real.fit -r12 >>R12
plot -dg,FProcs sys -r13 >R13
plot -ag,FProcs sys.fit -r13 >>R13
plot -dg,FProcs user -r8 >R8
plot -ag,FProcs user.fit -r8 >>R8
ged R12 R13 R8
```

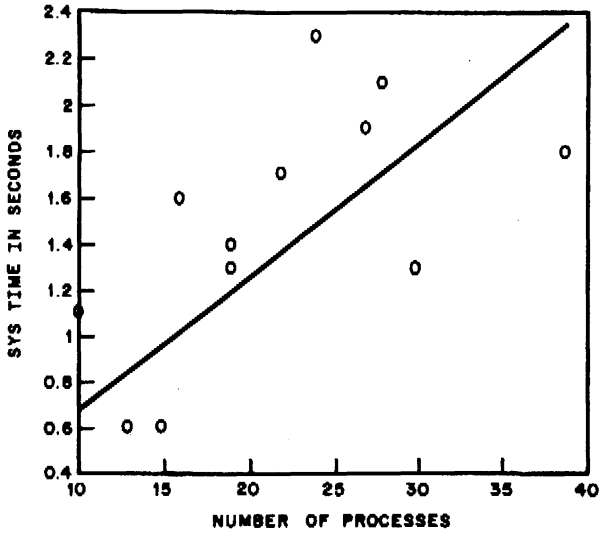
NOTES

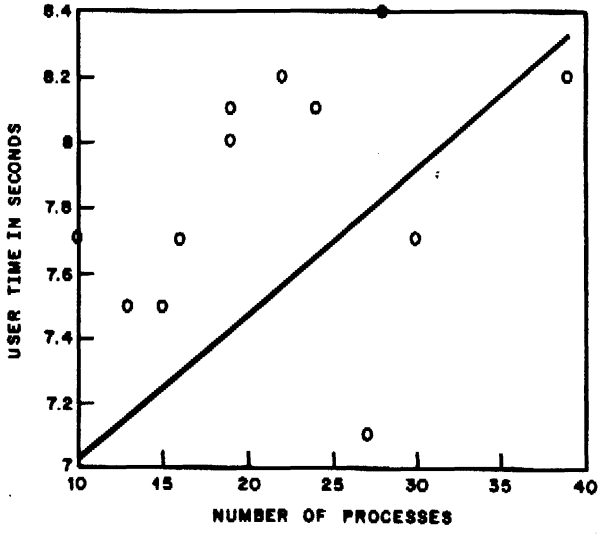
The performance data is the execution time, as reported by the UNIX **time** command, to generate the first 1000 prime numbers. **Time** outputs three times for each run: the time in system routines, the time in user routines, and the total real time. Each of these types of time is treated separately by the analysis program.

The short **awk** program converts "minutes:seconds" format to "seconds." **Lreg** does a linear regression of the time vectors on the size of the process table. **Siline** generates a line based on the parameters from the regression. One plot is generated for each type of time. Each plot is put into a different region so that they can be displayed and manipulated simultaneously in the graphical editor.

Figure a2


REAL TIME IN SECONDS
NUMBER OF PROCESSES


SYS TIME IN SECONDS
NUMBER OF PROCESSES


USER TIME IN SECONDS
NUMBER OF PROCESSES

Administrative Information for the UNIX Graphics Package

R. L. Chen
D. E. Pinkston
A. Guyton (4/1/80 revision)

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

This document is a reference guide for system administrators who are using or establishing a Graphics facility [1] on a UNIX† system. It contains information about directory structure, installation, makefiles, hardware requirements, and miscellaneous facilities of the Graphics Package.

2. GRAPHICS STRUCTURE

Figure 1 contains a graphical representation of the directory structure of Graphics. In this paper, the shell variable SRC will represent the parent node for Graphics source and is usually set **/usr/src/cmd**.

The *graphics* command (see *graphics*(1G)) resides in **/usr/bin**. All other Graphics executables are located in **/usr/bin/graf**; the **/usr/lib/graf** directory contains text for *whatis* documentation (see *gutil*(1G)) and editor scripts for *ttoc* (see *toc*(1G)).

Graphics source resides below the directory **$SRC/graf**; **$SRC/graf** is broken into the following subdirectories:

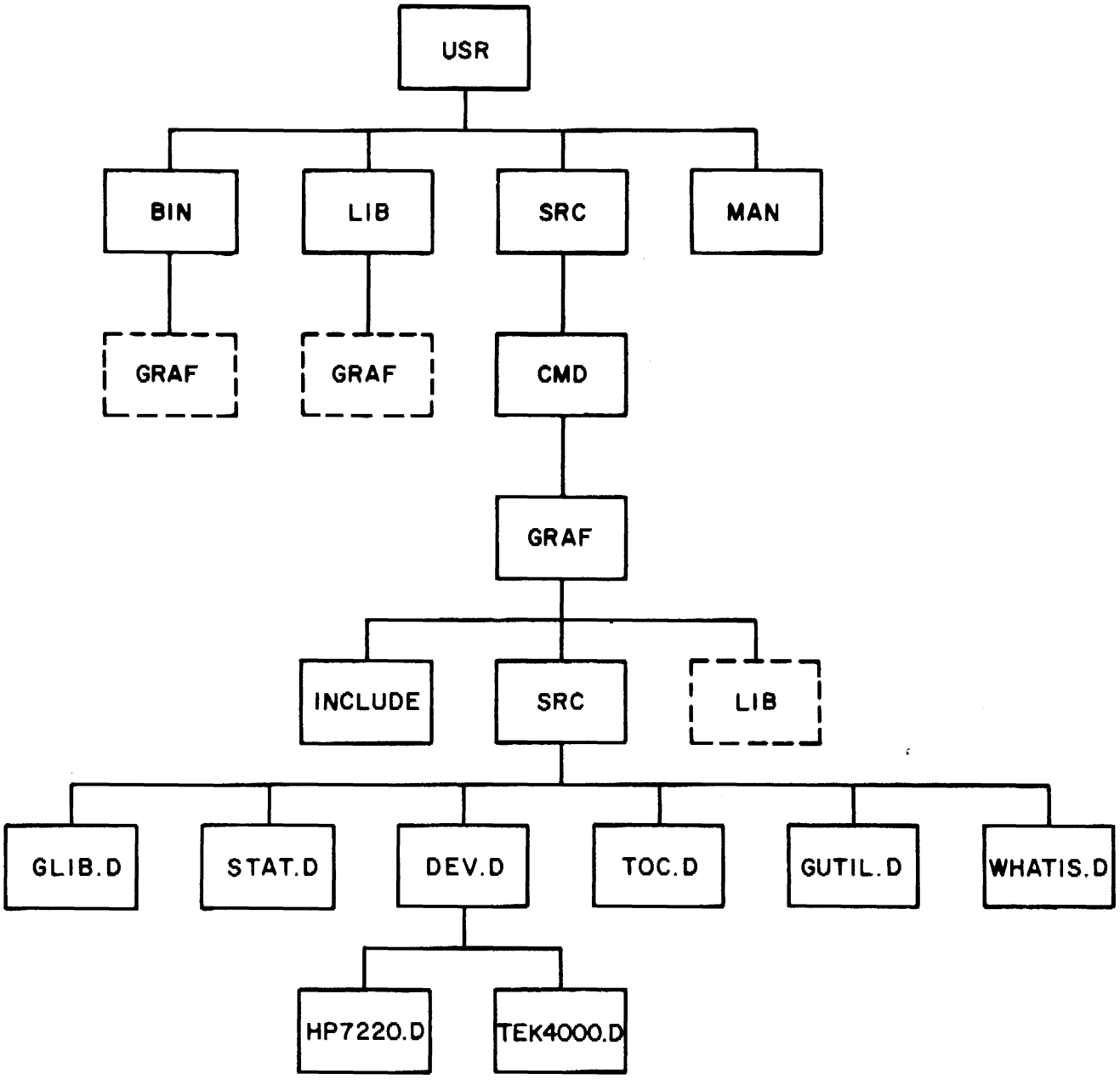
- **include** — contains the following header files: **debug.h**, **errpr.h**, **gsl.h**, **gpl.h**, **setopt.h**, and **util.h**.
- **src** — contains source code partitioned into subdirectories by subsystem. Each subdirectory contains its own *Makefile* (or *Install* file for **whatis.d**).
  - **glib.d** — contains source used to build the graphical subroutine library, **$SRC/graf/lib/glib.a**.
  - **stat.d** — contains source for numerical manipulation and plotting routines.
  - **dev.d** — contains source code for device filters partitioned into subdirectories.
    - **lolib** and **uplib** — contain source used to create device independent libraries.
    - **hp7220.d** — contains source for *hpd* (a Hewlett-Packard Plotter display function).
    - **tek4000.d** — contains source for *ged* (the graphical editor), *td* (a Tektronix display function), and other Tektronix dependent routines.
  - **gutil.d** — contains source for utility programs.
  - **toc.d** — contains source for table of contents drawing routines.
  - **whatis.d** — contains *nroff* files and the installation routine for on-line documentation.
- **lib** — contains **glib.a** which contains commonly used graphical subroutines.

UNIX User's Manual entries for Graphics consist of the following: *gdev*(1G), *ged*(1G), *graphics*(1G), *gutil*(1G), *stat*(1G), *toc*(1G), and *gps*(5).

---

† UNIX is a trademark of Bell Laboratories.

Figure 1. Graphics Directory Structure



3. INSTALLING GRAPHICS

Procedures for installing Graphics:

— To build the entire Graphics package, execute (as super-user):

/usr/src/:mkcmd graf

— To build a particular graphics subsystem use the shell variable ARGS:

ARGS=*subsystem* /usr/src/:mkcmd graf

A *subsystem* is either **glib**, **stat**, **dev**, **toc**, **gutil**, or **whatis**. **Glib** must exist before other subsystems can be built. Write permission in **/usr/bin** and **/usr/lib** is needed, and the following libraries are assumed to exist:

/lib/libc.a	Standard C library, used by all subsystems.
/lib/libm.a	Math library, used by all subsystems.
/usr/lib/macros/mm[nt]	Memorandum macros for [*nt*]*roff*, used by the *whatis* subsystem.

The complete build process takes approximately two hours of system time. If the build must be stopped, it is a good idea to restart from the beginning. Upon completion, the following things will be created and owned by **bin**:

/usr/lib/graf	A directory for data and editor scripts.
/usr/bin/graf	A directory for executables.
/usr/bin/graphics	Command entry point for Graphics.

Makefiles use executable shell procedures *cco* and *cca*. *Cco* is used to compile C source and install load modules in **/usr/bin/graf**. The *cca* command compiles C programs and loads object code into archive files.

Whatis.d contains source files for *whatis* and the executable command *Install*.

Install command-name

calls *nroff* to produce *whatis* documentation for *command-name* in **/usr/lib/graf**. To install the entire *whatis* subsystem, use *:mkcmd* as described above.

3.1 Makefile Parameters

Makefiles use various macro parameters, some of which can be specified on the command line to redirect outputs or inputs. Parameters specified in higher level *Makefiles* are passed to lower levels. Below is a list of specifiable parameters for *Makefiles* followed by their default values in parentheses and an explanation of their usage:

— $SRC/graf/graf.mk:

BIN (/usr/bin)	installation directory for the *graphics* command.
BIN (/usr/bin/graf)	installation directory for other graphic commands.
SRC (/usr/src/cmd)	parent directory for source code.

— $SRC/graf/src/Makefile

BIN1 (/usr/bin)	installation directory for the *graphics* command.
BIN2 (/usr/bin/graf)	installation directory for other graphic commands.
LIB (/usr/lib/graf)	installation directory for *whatis* documentation.

— $SRC/graf/src/stat.d/Makefile:

BIN (../../bin)	installation directory for executable commands.

— $SRC/graf/src/toc.d/Makefile:

BIN (../../bin) installation directory for executable commands.

— $SRC/graf/src/dev.d/Makefile:

BIN (../../bin) installation directory for executable commands.

— $SRC/graf/src/dev.d/hp7220.d/Makefile:

BIN (../../../bin) installation directory for executable commands.

— $SRC/graf/src/dev.d/tek4000.d/Makefile:

BIN (../../../bin) installation directory for executable commands.

— $SRC/graf/src/gutil.d/Makefile:

BIN (../../bin) installation directory for executable commands.

The following example will make a new version of the graphical editor, *ged*, installing it in **/a1/pmt/dp/bin**:

```
cd $SRC/graf/src/dev.d/tek4000.d
make BIN=/a1/pmt/dp/bin ged
```

(This assumes, of course, that necessary libraries were previously built.)

4. HEWLETT-PACKARD PLOTTER

The Graphics display function *hpd* uses the Hewlett-Packard 7221A Graphics Plotter. The HP plotter can be connected to the computer in series with a terminal via a dedicated or dial-up line. This arrangement allows the plotter to intercept plotting instructions while passing other data to the terminal unaltered and thus providing for normal terminal operation. Plotter switch settings should match those of the terminal. See the plotter operating manual for a more complete discussion [3].

5. TEKTRONIX TERMINAL

The Graphics display function *td* and the graphical editor *ged* both use Tektronix Series 4010 storage tubes. Below is a list of device considerations necessary for Graphics operation.

5.1 Inittab Entry

When a Tektronix 4010 series terminal is connected to UNIX via a dedicated 4800 or 9600 baud line, **/etc/inittab** should reference speed table entry **6** (may vary locally) of *getty*. Speed table entry **6** is designed specifically for the Tektronix 4014 and, among other things, sets a form-feed delay so that the screen may be cleared without losing information and clears the screen before prompting for a login. See *stty*(1), *inittab*(5) and *getty*(8) for more information.

5.2 Strap Options

The standard strap options as listed below should be used (see the Reference Manual for the Tektronix 4014 [2]):

— LF effect — LF causes line-feed only.
— CR effect — CR causes carriage return only.
— DEL implies loy — DEL key is interpreted as low-order y value.
— Graphics Input terminators — None.

5.3 Enhanced Graphics Module

The Enhanced Graphics Module (EGM) for Tektronix terminals is required for Graphics. The EGM provides different line styles (solid, dotted, dot-dashed, dashed, and long-dashed), right and left margin cursor location, and 12-bit cursor addressing (4096 by 4096 screen points).

6. MISCELLANEOUS INFORMATION

6.1 Announcements

The *graphics* command provides a means of printing out announcements to users. To set up an announcement facility, create a readable text file containing the announcements named **announce**. Also in **/usr/bin/graphics** redefine the shell variable GRAF to be the directory path name of the file **announce**.

6.2 Uselog

The *graphics* command also provides a means of monitoring its use by listing users in a file. To set up a usage logging facility, create a writable file named **.uselog** (in the same directory as **announce** if announcements are being used) and redefine the shell variable GRAF within **/usr/bin/graphics** to specify the directory location. Each time a user executes *graphics*, an entry of the login name, terminal number, and system date are recorded in **.uselog**.

6.3 Restricted Environments

Restricted environments can be used to limit access to the system (see *sh*(1)). A restricted environment for Graphics can be set up by creating the directories **/rbin** and **/usr/rbin** and populating them with restricted versions of regular UNIX commands, so that the environment cannot be compromised. In particular, *ed*(1), *mv*(1), *rm*(1), and *sh*(1) require restricted interface programs that do not allow users to move or remove files whose names begin with "." [4].

To create a restricted environment for Graphics:

— Create a restricted *ged* command in **/usr/rbin** as follows:

```
exec /usr/bin/graf/ged -R
```

— Create restricted logins for users or create a community login with a working directory (reached through **.profile**) set up for each user. A restricted login specifies **/bin/rsh** as the terminal interface program and is created by adding **/bin/rsh** to the end of the **/etc/passwd** file entry for that login.

— Call **graphics -r** from **.profile**.

The execution of **graphics -r** changes $PATH to look for commands in **/rbin** and **/usr/rbin** before **/bin** and **/usr/bin** and executes a restricted shell. The **-R** option is appended to the *ged* command so that the escape from *ged* to UNIX (*!command*) will also use a restricted shell.

ACKNOWLEDGEMENTS

We wish to thank A. R. Feuer for his valuable contributions, suggestions, and careful reading of this document. We also thank M. J. Petrella for his help in supplying information concerning the UNIX environment.

REFERENCES

[1] A. R. Feuer. *UNIX Graphics Overview*, Bell Laboratories (1979).
[2] *User's Manual for 4014 and 4014-1 Display Terminal*, Tektronix (July 1974).
[3] *7221A Graphics Plotter Operating and Programming Manual*, Hewlett-Packard (Nov. 1977).
[4] T. A. Dolotta, S. B. Olsson, and A. G. Petruccelli (eds.). *UNIX User's Manual*—Release 3.0, Bell Laboratories (June 1980).

January 1981

UNIX Remote Job Entry User's Guide

A. L. Sabsevitz
K. A. Kelleman

Bell Laboratories
Piscataway, New Jersey 08854

1. PREFACE

A set of background processes running under UNIX† support remote job entry to IBM System/360 and /370 host computers. RJE is the communal name for this subsystem.[1] UNIX communicates with IBM's Job Entry Subsystem by mimicking an IBM 360 remote multileaving work station. The *UNIX User's Manual* entry *rje*(8) summarizes its design and operation. The manual also contains a description of the *send*(1C) command, which is the user's primary method of submitting jobs to RJE, and *rjestat*(1C), which allows the user to monitor the status of RJE and to send operator commands to the host system. This guide is a tutorial overview of RJE and is addressed to the user who needs to know how to use the system, but does *not* need to know details of its implementation. The two following sections constitute an introduction to RJE.

2. PRELIMINARIES

To become a UNIX user, you must receive a login name that identifies you to the UNIX system. You should also get a copy of the *UNIX User's Manual;* it contains a fairly complete description of the system and includes the section *How to Get Started*, which introduces you to UNIX; you should read that section before proceeding with this guide.

In order to begin using RJE, you need only become familiar with a subset of basic commands. You must understand the directory structure of the file system, and you should know something about the attributes of files: see *cd*(1), *chmod*(1), *chown*(1), *cp*(1), *ln*(1), *ls*(1), *mkdir*(1), *mv*(1), *rm*(1). You must know how to enter, edit, and examine text files: see *cat*(1), *ed*(1), *pr*(1). You should know how to communicate with other users and with the system: see *mail*(1), *mesg*(1), *who*(1), *write*(1). And, finally, you might have to know how to describe your terminal to the system: see *ascii*(5), *stty*(1), *tabs*(1).

3. BASIC RJE

Let's suppose that you have used the editor, *ed*(1), to create the file, **jobfile**, that contains your job control statements (JCL) and input data. This file should look exactly like a card deck, except that for convenience alphabetic characters may be in either upper or lower case. Here is an example:

---

† UNIX is a trademark of Bell Laboratories.

1. In this paper, RJE refers to the facilities provided by UNIX, and *not* to the Remote Job Entry feature of IBM's HASP and JES subsystems.

```
$ cat jobfile
//gener job (9999,r740),pgmrname,class=x usr=(mylogin,myplace)
//step exec pgm=iebgener
//sysprint dd sysout=a
//sysin dd dummy
//sysut2 dd sysout=a
//sysut1 dd *
   first card of data
        :
        .
   last card of data
/*
```

To submit this job for execution, you must invoke the *send*(1C) command:

```
$ send jobfile
```

The system will reply:

```
10 cards
Queued as /usr/rje/rd3125
```

Note that *send* tells you the number of cards it submitted and reports the file name that contains your job in the queue of all jobs waiting to be transmitted to the host system. Until the transmission of the job actually begins, you can prevent the job from being transmitted by doing a **chmod 0** on the queued file to make it unreadable. For our example, you could say:

```
chmod 0 /usr/rje/rd3125
```

When your job is accepted by the host system, a job number will be assigned to it, and an acknowledgement message will be generated. This indicates that your job has been scheduled on the host system. Later, after the job has executed, its output will be returned to the UNIX system. You will be notified automatically of both of these events: if you are logged in when RJE detects these events, and if you are permitting messages to be sent to your terminal (see *mesg*(1)). The following two messages will be sent to you (still using the example above) when the job is scheduled and when the output is returned, respectively:

```
      Two bells
12:18:42 gener job 384 -- rd3125 acknowledged

      Two bells
12:21:54 gener job 384 -- /a1/user/rje/prnt0 ready
```

Two bells, with an interval of one second between them, precede each message. They should be interpreted as a warning to stop typing on your terminal, so that the imminent message is not interspersed with your typing.

If you are not logged in when one of these events occurs, or if you do not allow messages to be sent to your terminal, then the notification will be posted to you via the *mail*(1) command. You can prevent messages directly by executing the *mesg*(1) command, or indirectly by executing another command, such as *pr*(1), which prohibits messages for as long as it is active. You may inspect (by invoking the *mail* command) your mail file (/usr/mail/*logname*) at any time for messages that have been diverted. Setting your **MAIL** variable to the name of your mail file will cause the shell to notify you when mail arrives. For this example, the mail might look as follows:

```
$ mail
From rje Mon Aug  1 12:20:36 1977
12:18:42 gener job 384 -- rd3125 acknowledged

? d
From rje Mon Aug  1 12:21:55 1977
12:21:54 gener job 384 -- /a1/user/rje/prnt0 ready

? d
```

The job acknowledgement message performs two functions. First, it confirms the fact that your job has been scheduled for eventual execution. Second, it assigns a number to the job in such a way that the number and the name together will uniquely identify the job for some period of time.

The output ready message provides the name of a UNIX file into which output has been written and identifies the job to which the output belongs (see *ls*(1)):

```
$ ls -l prnt0
-r--r-xr--  1 rje          1184 Aug  1 12:21 prnt0
```

Note that rje retains ownership of the output and allows you only read access to it. It is intended that you will inspect the file, perhaps extract some information from it, and then promptly delete it (see *rm*(1)):

```
$ rm -f prnt0
```

The retention of machine-generated files, such as RJE output, is discouraged. It is your responsibility to remove files from your RJE directory. RJE output files may be truncated if the output exceeds a set limit. This limit is tunable by the system administrator. Output beyond the current limit will be discarded, with no provision for retrieval. If the output were truncated in the previous example, the second notification message would have been:

```
      Two bells
12:21:54 gener job 384 -- /a1/user/rje/prnt0 ready (truncated)
```

The user should also be aware that RJE attempts to keep a set number of blocks free on any file system it uses. This number is also tunable by the system administrator. Warning messages or suspension of certain functions will occur as this limit is approached.

The most elementary way to examine your output is to *cat* it to your terminal. The Appendix of this document shows the result of listing the output of our sample job in this way. Because UNIX has no high volume printing capability, you should route to the host's printer any large listings of which you desire a hard copy.

The structure of an output listing will generally conform to the following sequence:

```
HASP log
jcl information
data sets
HASP end
```

Normally burst pages will not be present. Single, double, and triple spacing is reflected in the output file, but other forms controls, such as the skip to the top of a new page, are suppressed. Page boundaries are indicated by the presence of a blank (space character) at the end of the last line of each page.

The big file scanner *bfs*(1) or the context editor *ed*(1) provide a more flexible method than *cat*(1) for examining printed output; *bfs* can handle files of any size and is more efficient than *ed* for scanning files.

RJE is also capable of receiving punched output as formatted files (see *pnch*(5)); this format allows an exact representation of an arbitrary card deck to be stored on the UNIX machine. However, there are few commands that can be used to manipulate these files. You will probably want to route your punched output to one of the host's output devices.

4. SEND COMMAND

The *send*(1C) command is capable of more general processing than has been indicated in the previous section. In the first place, it will concatenate a sequence of files to create a single job stream. This allows files of JCL and files of data to be maintained separately on the UNIX machine. In addition, it recognizes any line of an input file that begins with the character ~ as being a *control* line that can call for the inclusion, inside the current file, of some other file. This allows you to *send* a top level skeleton that "pulls" in subordinate files as needed. Some of these may be "virtual" files that actually consist of the output of UNIX commands or Shell procedures. Furthermore, the *send* command is able to collect input directly from a terminal, and can be instructed to prompt for required information.

Each source of input can contain a format specification that determines such things as how to expand tabs and how long can an input line be. The manual entry for *fspec*(5) explains how to define such formats. When properly instructed, *send* will also replace arbitrarily defined keywords by other text strings or by EBCDIC character codes. (These two substitution facilities are useful in other applications besides RJE; for that reason, *send* may be invoked under the name *gath* to produce standard output *without* submitting an RJE job.)

Two options of *send* that everyone should be acquainted with are: the ability to specify to which host computer the job is to be submitted, and a flag that guarantees that a job will be transmitted to the host computer in order of submission (relative to other jobs submitted with the same flag). To run our sample job on a host machine known to RJE as A, we would issue the command:

```
$ send A jobfile
```

When no host is explicitly cited, *send* makes a reasonable choice.

To insure that a job will be transmitted in order of submission, set the **−x** flag:

```
$ send −x jobfile
```

This flag should be used sparingly. The complete list of arguments and flags that control the execution of *send* can be found in *send*(1C).

5. JOB STREAM

It is assumed that the job stream submitted as the result of a single execution of *send* consists of a single *job*, i.e., the file that is queued for transmission should contain one JOB card near the beginning and no others. A priority control card may legitimately precede the JOB card. The JOB card must conform to the local installation's standard. At BISP, it has the following structure:

```
//name job (acct[,...]),pgmrname[,keywds=?] [usr=...]
```

6. USER SPECIFICATION

A "usr" specification is required on print or punch output that is to be delivered to a UNIX user.

```
usr=(login,place,[level])
```

where *login* is the UNIX login name of the user, *level* is the desired level of notification (see end of this section for an explanation), and *place* is as follows:

A. If *place* is the name of a directory (writable by others), then the output file is placed there as a unique **prnt** or **pnch** file. The mode of the file will be 454.

B. If *place* is the name of an existing, writable (by others), non-executable (by others) file, then the output file replaces it. The mode of the file will be 454.

C. If *place* is the name of a non-existent file in a writable (by others) directory, then the output file is placed there. The mode of the file will be 454.

D. If *place* is the name of an executable (by others) file, then the RJE output is set up as standard input to *place*, and *place* is executed. Five string arguments are passed to *place*. For example, if *place* is a shell procedure, the following arguments are passed as $1 ... $5:

1. Flag indicating whether file space is scarce in the file system where *place* resides. A **0** indicates that space is *not* scarce, while **1** indicates that it is.
2. Job name.
3. Programmer's name.
4. Job number.
5. Login name from the "usr=..." specification.

A ":" is passed if a value is not present. The current directory for the execution of *place* will be set to the directory containing *place*. The environment (see *environ*(7)) will contain values for **LOGNAME** and **HOME** based on the login name from the "usr=..." specification, and a value for **TZ**. Since the login name supplied on the "usr=..." specification cannot be believed for security purposes, the UID will be set to a reserved value.

E. In all other cases, the output will be thrown away.

The *place* value must not be a full path name, unless it refers to an executable file (see D above). For cases A, B, and C above (and case D, if a full path name is not supplied), the name of the user's login directory will be used to form a full path name.

The "usr=..." field may occur anywhere within the first 100 card images sent and within the first 200 output images received by the UNIX system. The only restriction is that it be contained completely on a single line or card image. Therefore, the "usr=..." field may be placed on a JOB card or comment card. It may also be passed as data.

For redirection of output by the host, a "usr=..." card, if not already present, must be supplied by the user. This can be done by placing a job step that creates this card before your output steps.

Messages generated by RJE or passed on from the host are assigned a level of importance ranging from 1 to 9. The levels currently in use are:

3 transmittal assurance
5 job acknowledgement
6 output ready message

The optional *level* field of the "usr=..." specification must be a one or two-digit code of the form *mw*. A message from the host with importance x (where x comes from the above list) is compared with each of the two decimal digits in *level*. If $x \geq w$ and if the user is logged in and is accepting messages, the message will be written to his or her terminal. Otherwise, if $x \geq m$, the message will be mailed to the user. In all other cases, the message will be discarded. The default *level* is **54**. You should specify level **1** if you want to receive complete notification, and level **59** to divert the last three messages in the above list to your mailbox.

7. MONITORING RJE

RJE is designed to be an autonomous facility that does not require manual supervision. RJE is initiated automatically by the UNIX reboot procedures and continues in execution until the system is shut down. Experience has shown RJE to be reasonably robust, although it is vulnerable to system crashes and reconfigurations.

Users have a right to assume that when the UNIX system is up for production use, RJE will also be up. This implies more than an ability to execute the *send*(1C) command, which should be available at all times; it means that queued jobs should be submitted to the host for execution and their output returned to the UNIX system. If a user cannot obtain any throughput from RJE, he or she should so advise the UNIX operators.

The *rjestat*(1C) command, invoked with no arguments will report the status of all RJE links for which a given UNIX system is configured. It may sometimes also print a message of the day from RJE.

```
$ rjestat
RJE to B operating normally.
RJE to A down, reason: IBM not responding.
```

A host machine may be reported to be not responding to RJE because it is down, or because of its operator's failure to initialize the associated line, or because of a communications hardware failure.

Rjestat also has the ability to send operator commands to the host machine and retrieve the responses generated by the commands. Refer to the *rjestat*(1C) manual entry for a complete description of this command.

APPENDIX

Sample JES2 Output Listing

```
$ cat rje/prnt0
14.40.31 JOB 384  $HASP373 GENER  STARTED - INIT 26 - CLASS X - SYS RRMA
14.40.32 JOB 384  $HASP395 GENER  ENDED

------ JES2 JOB STATISTICS ------

 1 AUG 77 JOB EXECUTION DATE

          54 CARDS READ

          76 SYSOUT PRINT RECORDS

           0 SYSOUT PUNCH RECORDS

      0.01 MINUTES EXECUTION TIME
   1        //GENER JOB (9999,R740),PGMRNAME,CLASS=X                    JOB 384
            ***       USR=(MYLOGIN,MYPLACE)
   2        //IEBGENER EXEC PGM=IEBGENER
   3        //SYSPRINT DD DUMMY
   4        //SYSIN DD DUMMY
   5        //SYSUT2 DD SYSOUT=A
   6        //SYSUT1 DD *
            //
IEF236I ALLOC. FOR GENER IEBGENER
IEF237I DMY ALLOCATED TO SYSPRINT
IEF237I DMY ALLOCATED TO SYSIN
IEF237I JES ALLOCATED TO SYSUT2
IEF237I JES ALLOCATED TO SYSUT1
IEF142I GENER IEBGENER - STEP WAS EXECUTED - COND CODE 0000
IEF285I   JES2.JOB0384.SO0102                          SYSOUT
IEF285I   JES2.JOB0384.SI0101                          SYSIN
IEF373I STEP /IEBGENER/ START 77242.1440
IEF374I STEP /IEBGENER/ STOP 77242.1440 CPU 0MIN 00.13SEC SRB 0MIN 00.01SEC VIRT 36K SYS 188K

******* SERVICE UNITS=0000174          SERVICE RATE=0000268 SERVICE UNITS/SECOND
******* PERFORMANCE GROUP=005
******* EXCP COUNT BY UNIT ADDRESS
IEF375I  JOB /GENER / START 77242.1440
IEF376I  JOB /GENER / STOP  77242.1440 CPU    0MIN 00.13SEC SRB    0MIN 00.01SEC

******* SERVICE UNITS=0000174          SERVICE RATE=0000268 SERVICE UNITS/SECOND
******* APPROXIMATE PROCESSING TIME=           .01 MINUTES
******* EXCPS=000000000
******* PROJECTED CHARGES=             .01

          first line of data
            :
            :
          last line of data

*OS/VS2 REL 3.7 JES2* END  JOBNAME=GENER   BIN=R740      JOB #=384  PGMRNAME
*OS/VS2 REL 3.7 JES2* END  JOBNAME=GENER   BIN=R740      JOB #=384  PGMRNAME
*OS/VS2 REL 3.7 JES2* END  JOBNAME=GENER   BIN=R740      JOB #=384  PGMRNAME

$ rm -f rje/prnt0
```

UNIX Remote Job Entry Administrator's Guide

M. J. Fitton

Bell Laboratories

1. INTRODUCTION

1.1 Purpose

This document is intended to augment the existing body of documentation on the design and operation of UNIX† IBM RJE[1]. The reader should be familiar with *rje*(8), and the *UNIX Remote Job Entry User's Guide*, April 1, 1980. There will be assumptions made concerning allocation of responsibilities between UNIX and IBM operations, hardware configuration, etc. Although these assumptions may not fully apply to your location, they should not interfere with the intent of this document.

The major topics discussed in this paper are as follows:

- SETTING UP — hardware requirements and RJE generation on the IBM and UNIX systems.
- DIRECTORY STRUCTURES — the controlling RJE directory structure and a typical RJE subsystem directory structure.
- RJE PROGRAMS — programs that make up an RJE subsystem.
- UTILITY PROGRAMS — utility programs that are available for debugging or tracing.
- RJE ACCOUNTING — the accounting of jobs done by RJE, and some methods for using this accounting data.
- TROUBLE SHOOTING — error recovery and procedures for identifying and fixing RJE problems.

1.2 Facilities

Discussions will focus on a hypothetical RJE connection between a UNIX system, *pwba*, and an IBM 370/168, referred to as *B*. We also assume that *pwba* is connected to an IBM 370/158, referred to as *C*. The UNIX machine emulates an IBM System/360 remote multi-leaving work station. For more information on the multi-leaving protocol, see Appendix B of *OS/VS MVS JES2 Logic* (SY24-6000-1).

2. SETTING UP

2.1 Hardware

To use RJE on a UNIX system the following hardware is needed (one per remote line):

- KMC11-B Microprocessor — used to drive the RJE line
- DMC11-DA or DMC11-FA line unit — the DMC11-DA interfaces with Bell 208 and 209 synchronous modems or equivalent. Speeds of up to 19,200 bits per second can be used. The DMC11-FA interfaces with Bell 500 A LI/5 synchronous modems or equivalent. Speeds of up to 250,000 bits per second can be used.

---

† UNIX is a trademark of Bell Laboratories.

1. In this paper, RJE refers to the facilities provided by UNIX and *not* to the Remote Job Entry feature of IBM's HASP and JES2 subsystems.

On the DMC11 line unit, the Cyclic Redundancy Check (CRC) switch should be set to inhibit automatic transmission of CRC bytes. The line unit should hold the line at logical zero when inactive. For a more detailed description of the above hardware, see *Terminals and Communications Handbook*, Digital Equipment Corporation, 1979.

2.2 IBM Generation

The following applies to the host IBM system. The remote line to the UNIX machine should be described as a System/360 remote work station. The following parameters must be initialized and *must* agree with their counterparts on the UNIX machine:

- Number of printers (NUMPR) – the number of logical printers (up to 7)
- Number of punches (NUMPU) – the number of logical punches (up to 7)
- Number of readers (NUMRD) – the number of logical readers (up to 7)

The JES2 parameters for our hypothetical connection to IBM system *B* are as follows:

```
RMT5 S/360,LINE=5,CONSOLE,MULTI,TRANSP,NUMPR=5,
        NUMPU=1,NUMRD=5,ROUTECDE=5
R5.PR1 PRWIDTH=132
R5.PR2 PRWIDTH=132
R5.PR3 PRWIDTH=132
R5.PR4 PRWIDTH=132
R5.PR5 PRWIDTH=132
R5.PU1 NOSUSPND
R5.RD1 PRIOINC=0,PRIOLIM=14
R5.RD2 PRIOINC=0,PRIOLIM=14
R5.RD3 PRIOINC=0,PRIOLIM=14
R5.RD4 PRIOINC=0,PRIOLIM=14
R5.RD5 PRIOINC=0,PRIOLIM=14
```

System *pwba* is referenced by line 5 (LINE=5), remote 5 (RMT5). It is defined as having a console, for the *rjestat*(1C) command, five printers, one punch, and five readers. Although you may have up to seven printers or punches, the total number of printers and punches may not exceed eight. The line is described as a transparent (TRANSP), multi-leaving (MULTI) line. The remaining information describes attributes of the printers, punches, and readers.

Normally, separator pages are transmitted with IBM print files. UNIX RJE does not remove separator pages. To prevent transmission of separator pages on printer 1 of the previous example, its attributes would be:

```
R5.PR1 PRWIDTH=132,NOSEP
```

NOSEP should be included for all printers when separator pages are not desired. Most IBM systems can also be told via a console command to cancel transmission of separator pages on printers. This can be done from the IBM system console, or from the remote UNIX machine via *rjestat*. For example, the following JES2 command would cancel separator page transmission on printer 1:

```
$TR5.PR1,S=N
```

2.3 UNIX Generation

If the RJE remote dialing facility is to be used, the administrator must make sure that the definition for RJECU in the file **/usr/include/rje.h** is the device to be used for remote dialing. RJECU is defined to be **/dev/dn2** when distributed. To compile and install RJE, the normal *make*(1) procedures are used (see *Setting up UNIX*). Once an RJE subsystem has been installed, the remote line must be described in the configuration file **/usr/rje/lines**. This file as it exists on our hypothetical system *pwba* is as follows:

```
B pwba /usr/rje1 rje1 vpm0 5:5:1 1200:512:y
C pwba /usr/rje2 rje2 vpm1 1:1:1 1200:512
```

/usr/rje/lines is accessed by all components of RJE. Each line of the table (maximum of 8) defines an RJE connection. Its seven columns may be labeled **host**, **system**, **directory**, **prefix**, **device**, **peripherals**, and **parameters**. These columns are described as follows:

- **host** — The IBM System name, e.g., **A**, **B**, **C**. This string can be up to 5 characters long.
- **system** — The UNIX System name (see *uname*(1)).
- **directory** — the directory name of the servicing RJE subsystem (e.g., **/usr/rje2**).
- **prefix** — the string prepended to most files and programs in the **directory** (i.e., **rje2**).
- **device** — the name of the controlling Virtual Protocol Machine (VPM) device, with **/dev/** excised. In order to specify a VPM device, all VPM software must be installed, and the proper special files must be made (see *vpm*(4) and *mknod*(1M)).
- **peripherals** — information on the logical devices (readers, printers, punches) used by RJE. There are three subfields. Each subfield is separated by ":" and is described as follows:

  1. Number of logical readers.
  2. Number of logical printers.
  3. Number of logical punches.

  Note: the number of peripherals specified for an RJE subsystem *must* agree with the number of peripherals that have been described on the remote machine for that line.

- **parameters** — this field contains information on the type of connection to make. Each subfield is separated by ":". Any or all fields may be omitted; however, the fields are positional. All but trailing delimiters must be present. For example, in:

  1200:512:::9-555-1212

  subfields 3 and 4 are missing. Each subfield is defined as follows:

  1. **space** — this subfield specifies the amount of space (*S*) in blocks that RJE tries to maintain on file systems it touches. The default is 0 blocks. *Send*(1C) will not submit jobs and *rjeinit* issues a warning when less than 1.5*S* blocks are available; *rjerecv* stops accepting output from the host when the capacity falls to *S* blocks; RJE becomes dormant, until conditions improve. If the space on the file system specified by the user on the "usr=" card would be depleted to a point below *S*, the file will be put in the **job** subdirectory of the connection's home directory rather than in the place that the user requested.
  2. **size** — this subfield specifies the size in blocks of the largest file that can be accepted from the host without truncation taking place. The default is no truncation. Note that UNIX has a default one Mega-byte file size limit.
  3. **badjobs** — this subfield specifies what to do with undeliverable returning jobs. If an output file is undeliverable for any reason other than file system space limitations (e.g., missing or invalid "usr=" card) and this subfield contains the letter **y**, the output will be retained in the **job** subdirectory of the home directory, and login **rje** is notified via *mail*(1). If this subfield has any other value, undeliverable output will be discarded. The default is **n**.
  4. **console** — this subfield specifies the status of the interactive status terminal for this line. If the subfield contains an **i**, the status console facilities of *rjestat* will be inhibited. In all cases, the normal non-interactive uses of *rjestat* will continue to function. The default is **y**.

5. **dial-up** — this subfield contains a telephone number to be used to call a host machine. The telephone number may contain the digits 0 through 9, and the character "−", which denotes a pause. If the telephone number is not present, no dialing is attempted, and a leased line is assumed.

When multiple readers have been specified, jobs that are submitted for transmission to IBM are assigned to the reader with the fewest cards on it. Each reader gets an equal amount of service. This prevents smaller jobs from having to wait for a previously submitted large job to be transmitted. When multiple printers or punches have been specified, returning jobs get assigned to free printers (or punches) allowing smaller output files to bypass large output files.

Deciding how many peripherals to specify depends on the use of that RJE subsystem. If an RJE subsystem is heavily used for off-line printing (i.e., output does not return to the UNIX machine), the administrator would want to specify multiple readers, but would not have a need for multiple printers or punches.

3. DIRECTORY STRUCTURES

3.1 Controlling Directory

The controlling directory used by RJE is **/usr/rje**. This directory contains RJE programs for use by separate RJE subsystems (e.g., **rje1**, **rje2**, **rje3**), and the shell queuer's directory. Most RJE programs existing here have been compiled such that each RJE subsystem shares the text of these programs. A snapshot of this directory on our hypothetical machine is as follows:

```
-rwxr-xr-x   2 rje      rje        4068 Mar  4 10:42 cvt
-rw-r--r--   1 rje      rje          42 Apr 10 09:52 lines
-rwxr-xr-x   2 rje      rje       15096 Apr 10 13:01 rjedisp
-rwxr-xr-x   2 rje      rje        2328 Mar  4 10:21 rjehalt
-rwxr-xr-x   2 rje      rje       10396 Apr 15 10:07 rjeinit
-r-x------   2 rje      rje         785 Apr  8 09:00 rjeload
-rwsr-xr-x   2 rje      rje        5040 Mar 27 09:28 rjeqer
-rwxr-xr-x   2 rje      rje        4072 Apr  1 15:40 rjerecv
-rwxr-xr-x   2 rje      rje        3888 Mar 27 09:35 rjexmit
-rwsr-xr-x   1 root     rje        2696 Mar 27 14:42 shqer
-rwxr-xr-x   2 rje      rje        5920 Apr  2 15:47 snoop
drwxr-xr-x   2 rje      rje          80 Mar 25 13:26 sque
```

RJE subsystems are generated in their own directory by linking the program names in this directory to the appropriate names in the subsystem directory. The programs are described in Section 4. The file **lines** is the configuration file used by all RJE subsystems. The directory **sque** is used by the Shell queuer (*shqer*). This directory contains:

```
-rw-r--r--   1 rje      rje           0 Feb 14 14:04 errors
-rw-r--r--   1 rje      rje           0 Feb 14 14:04 log
```

When *shqer* has work to do, the files **log** and **errors** will be of non-zero length, and temporary files (**tmp***) will also appear here. For a complete description of *shqer* and these files, see Section 4.8.

3.2 Subsystem Directory

The RJE subsystem described in this section maintains the connection between *pwba* and IBM *B*, and will be referred to as *rje1*. The first line of **/usr/rje/lines** (see Section 2.3) describes *rje1*. As noted in this file, *rje1* runs in the directory **/usr/rje1**. A snapshot of this directory is as follows:

```
-rw-r--r--   1 rje      rje        4990 Apr 15 08:30 acctlog
-rwxr-xr-x   2 rje      rje        4068 Mar  4 10:42 cvt
-rw-r--r--   1 rje      rje           0 Apr 15 04:02 errlog
drwxrwxrwx   2 rje      rje         192 Apr 10 09:51 job
-rw-r--r--   1 rje      rje         194 Apr 15 08:11 joblog
-rw-r--r--   1 rje      rje           0 Apr 15 08:11 resp
-rwxr-xr-x   2 rje      rje       15096 Apr 10 13:01 rjeldisp
-rwxr-xr-x   2 rje      rje        2328 Mar  4 10:21 rjelhalt
-rwxr-xr-x   2 rje      rje       10396 Apr 15 10:07 rjelinit
-r-x------   2 rje      rje         785 Apr  8 09:00 rjelload
-rwsr-xr-x   2 rje      rje        5040 Mar 27 09:28 rjelqer
-rwxr-xr-x   2 rje      rje        4072 Apr  1 15:40 rjelrecv
-rwxr-xr-x   2 rje      rje        3888 Mar 27 09:35 rjelxmit
drwxr-xr-x   2 rje      rje         144 Apr 15 08:30 rpool
-rwxr-xr-x   2 rje      rje        5920 Apr  2 15:47 snoop0
drwxrwxrwx   2 rje      rje         176 Apr 10 13:03 spool
drwxr-xr-x   2 rje      rje         224 Apr 10 13:56 squeue
-rw-r--r--   1 rje      rje           0 Apr 15 10:30 stop
-rw-r--r--   1 rje      rje         274 Mar  7 20:25 testjob
```

The programs *rjel**, *cvt*, and *snoop0* are linked to the corresponding programs in **/usr/rje**, and are described in detail in Section 4. The remaining files and their uses are as follows:

- **acctlog** — accounting data is stored in this file, if it exists. This file is the responsibility of the RJE administrator. For a discussion of its uses, see Section 5.
- **errlog** — used by *rjel* to log errors. It can be useful for debugging *rjel* problems.
- **joblog** — used by *rjelqer* and *rjestat* to notify *rjelxmit* that a job (or console request) has been submitted. It also contains the process-group number of the *rjel* processes. The program *cvt* can be used to convert this file to a readable form.
- **resp** — contains console messages received from IBM *B*. These messages can be responses for *rjestat*, or IBM responses to submitted jobs (i.e., on reader messages). This file is truncated if it grows to a size greater than 70,000 bytes.
- **stop** — indicates that *rjelhalt* has been executed. The existence of this file indicates to *rjestat* that *rjel* has been halted by the operator.
- **testjob** — a sample job that can be submitted to test the *rjel* subsystem. Originally, the job control statements may have to be changed to suit your IBM system.

When *rjel* terminates abnormally, the file **dead** should appear in this directory. This file contains a short message indicating why *rjel* is not operating, and is used by *rjestat* to report the problem. The remaining directories and their uses are as follows:

- **job** — used to save undeliverable jobs, if the proper parameter has been specified in **/usr/rje/lines**. The sample job described above is also delivered to this directory. This directory should be mode 777.
- **rpool** — contains temporary files used to gather output from the remote machine. These files are named **pr*** (for print output files), and **pu*** (for punch output files). Once a complete file has been received, the file is dispatched in the proper way by *rjeldisp*.
- **spool** — used by *send* to store temporary files to be submitted to the remote machine. This directory must be mode 777.
- **squeue** — used by *rjel* to store submitted files until they are transmitted. The program *rjelqer* is used by *send* to move the temporary files in the **spool** directory to this directory.

4. RJE PROGRAMS

All programs described below, with the exception of *rjestat*, exist in **/usr/rje**. These programs are "shared text" and are linked (except *shqer*) to the proper names in each subsystem directory. The names described below are generic; the programs in the *rje2* directory would be *rje2qer*, *rje2init*, etc.

Each available RJE subsystem occupies three process slots. The slots are used for *rje?xmit*, the transmitter; *rje?recv*, the receiver; and *rje?disp*, the dispatcher. One additional process slot is used for *shqer*, regardless of how many subsystems are available.

Each RJE subsystem tries to be self-sustaining, and logs any errors encountered during normal operation in its **errlog** file.

4.1 Rjeqer

This program is used by *send* to queue files for transmission. When invoked, it performs the following steps:

1. Moves the temporary *pnch*(5) format file in the **spool** directory to the **squeue** directory.
2. Writes an entry at the end of the file **joblog** containing:
   - the name of the file to be transmitted
   - the submitter's user ID
   - the number of card images in the file
   - the message level for this job

   The file **joblog** is used to notify *rjexmit* of work to be done.
3. Notifies user that file has been queued.

Send determines which host system is desired, and invokes the proper *rje?qer* by getting the **prefix** from the **lines** file (e.g., if sending to IBM *C* from our machine, *rje2qer* would be invoked).

4.2 Rjeload

This program is used to start an RJE subsystem. Its prefix determines which subsystem to start (e.g., *rje2load* starts *rje2*). To start the RJE subsystems on our machine, the following commands are executed in **/etc/rc** when changing to *init* state 2 (multi-user):

```
rm -f /usr/rje/sque/log
su rje -c "/usr/rje1/rje1load vpb0 kmc0"
su rje -c "/usr/rje2/rje2load vpb1 kmc1"
```

The file **/usr/rje/sque/log** is removed to ensure the correct operation of *shqer*. When invoked, *rjeload* performs the following steps:

1. Uses the VPM device from **/usr/rje/lines** to link the proper devices (see *vpmset*(1C)).
2. Uses *kasb*(1) to perform the following:
   - reset the KMC
   - load the VPM script (**/etc/rjeproto**)
   - start the KMC running
3. Executes *rje?init* to start the *rje?* processes (e.g., *rje2load* executes *rje2init*).

4.3 Rjehalt

This program is used to halt an RJE subsystem. To halt *rje2* on our machine, **/usr/rje2/rje2halt** is executed. This should be done in the *shutdown* procedure for your machine to ensure graceful termination of RJE. *Rjehalt* will allow only those users with permission to halt an RJE subsystem. *Rjehalt* uses the header on the file **joblog** to get the process-group of the RJE subsystem processes. This group is signaled to terminate. When all processes have terminated, *rjehalt* sends a "signoff" record to the host machine. This signoff record is taken from the file **signoff** (ASCII text), if it exists, otherwise a "/*signoff" record is sent. On completion, *rjehalt* creates the file **stop** in the subsystem directory, that causes *rjestat* to report that RJE to the corresponding host has been stopped by the operator.

4.4 Rjeinit

This program initializes an RJE subsystem. It is used by *rjeload*, and can be used to restart a subsystem if the VPM script has previously been started. *Rjeinit* should only be executed by user **rje**. *Rjeinit* fails if there are less than 100 blocks or 10 inodes free in the file system. It issues a warning if there are less than 1.5X blocks, (where X is the first field in the parameters for that line), or 100 inodes free in the file system. If *rjeinit* fails, the reason for the failure is reported, and the file **dead** is created containing "Init failed". This will be reported by *rjestat* until a subsequent *rjeinit* succeeds. *Rjeinit* performs the following functions:

1. Dials a remote host if specified (see Section 2.3).
2. Truncates the console response file **resp**.
3. Sends a signon record to the host. The signon record is taken from the file **signon** (ASCII text), if it exists, otherwise *rjeinit* sends a blank record as a signon.
4. Sets up pipes for process communication.
5. Resets process-group for RJE subsystem and restarts error logging.
6. Rebuilds the **joblog** file from jobs queued for transmission.
7. Notifies *rjedisp* (via a pipe) of any returned files still remaining in the **rpool** directory.
8. Starts the appropriate background processes (*rje?xmit*, *rje?recv*, and *rje?disp*).
9. Reports started or not started.

If failure occurs in a background process, it is reported by that process (error logging). The failing process will normally attempt to reboot the subsystem by executing *rje?init* with a + as its argument (see Section 7). When *rjeinit* is executed with + as its argument, this indicates an attempted reboot, and *rjeinit* will behave differently (no re-dialing is done to remote hosts, errors are logged rather than printed, etc.).

4.5 Rjexmit

This program writes data to the VPM device. *Rjexmit* is started by *rjeinit* and runs in the background. When running, *rjexmit* performs the following processing:

1. Checks the **joblog** file for files to be transmitted. This is done every 5 seconds when not transmitting data. When transmitting data, the **joblog** is checked after transmitting 1 block from each active **reader**[2], and the **console**[3].

---

2. **Reader** refers to the logical readers used by RJE.
3. **Console** refers to the RJE logical console, which is separate from the logical readers.

2. Queues files from the **joblog** according to the first two characters of the file name:
   - **rd*** — these files are queued on the reader with the fewest cards. Normal use of the *send* command creates these files.
   - **sq*** — these files are queued on the last available reader to assure sequential transmission. Using the **−x** option to the *send* command creates these files.
   - **co*** — these files are queued on the console. The *rjestat* command creates these files.

   All files described above contain EBCDIC data.
3. Sends information to *rjedisp* (via a pipe) for use in user notification of job status (see Section 4.7).
4. Builds blocks for transmission from active readers and the console. These blocks are built according to the multi-leaving protocol.
5. Performs the following peripheral control:
   - Sends requests to open readers when jobs have been assigned to them. These readers are not active until a grant is received from *rjerecv* (via a pipe).
   - Halts or activates readers when waits or starts, respectively, are received from *rjerecv*.
   - Sends printer or punch grants when an open request is received from *rjerecv*.
6. Notifies *rjedisp* that a file has been transmitted, and unlinks the file.

If *rjexmit* encounters fatal errors, it creates the **dead** file with an appropriate message, and signals the other background processes to exit. If possible, *rjexmit* will attempt to reboot the RJE subsystem by executing *rjeinit*.

4.6 Rjerecv

This program reads data from the VPM device. *Rjerecv* is started by *rjeinit* and runs in the background. When running, *rjerecv* performs the following processing:

1. Reads blocks of data received from the host system.
2. Handles data received according to its type. The two types of data are:
   - **Control information** — *rjerecv* performs the following peripheral device control:
     a. Notifies *rjexmit* of grants to its requests to open readers.
     b. Passes wait and start reader information to *rjexmit*.
     c. Passes open requests (for printers and punches) from the host to *rjexmit*.
   - **User Information** — the three major types of user information received are:
     a. Console responses and job status messages. This data is appended to the **resp** file for use by *rjestat* and *rjedisp*.
     b. The printer output from user jobs. This data is collected in temporary files (**pr***) in the **rpool** directory. When a complete print job has been received, *rjerecv* notifies *rjedisp* (via a pipe) that the file is to be dispatched.
     c. The punch output from user jobs. This data is handled the same as printer output except that the **rpool** files are named **pu***.
3. If the console response file **resp** exceeds 70,000 characters, *rjerecv* truncates the file.
4. *Rjerecv* stops accepting output from the remote machine if the number of free blocks in the file system falls below **space** blocks (**space** is described in Section 2.3).

5. *Rjerecv* truncates received files to **size** blocks (**size** is described in Section 2.3).

If *rjerecv* encounters fatal errors, it creates the **dead** file with an appropriate error message, signals the other background processes to exit, and reboots the RJE subsystem.

4.7 Rjedisp

This program dispatches user information. *Rjedisp* is started by *rjeinit* and runs in the background. When running, *rjedisp* performs the following processing:

1. Dispatches output; the two types of output are printer and punch output. After receiving notification of output ready from *rjerecv*, *rjedisp* searches for a "usr=" line in the received file. The format of a "usr=" line is as follows:

   usr=(user,place,level)

   Rjedisp dispatches the output according to the place field. See *UNIX Remote Job Entry User's Guide* for a detailed description of the user specification.

2. Dispatches messages. The three types of messages are as follows:

   - Job transmitted — this message is sent to the submitting user when *rjedisp* reads this event notice from the *rjexmit* pipe.
   - Job acknowledgement — *rjedisp* dispatches IBM acknowledgement messages to submitting users. If a job is not acknowledged properly or within a reasonable amount of time, a "Job not acknowledged" message is dispatched.
   - Output processing — *rjedisp* dispatches job output messages according to the options specified on the "usr=" card. A normal output message indicates the returned file name is ready.

   Messages can be masked by using the *level* on the "usr=" card.

3. Whenever output is to be handled by *shqer*, *rjedisp* checks that *shqer* is running. This is done by looking for the *shqer* **log** file. If this file does not exist, *rjedisp* starts *shqer*.

4.8 Shqer

This program executes user programs when they appear in the *place* field of the "usr=" line in a returned output file (print or punch). *Shqer* is started by *rjedisp* when the first output file using this feature is returned. Subsequent files using this feature are logged for execution by *rjedisp*. When started, *shqer* performs the following processing:

1. Builds the **log** file from file names in the **/usr/rje/sque** directory. Each log entry is the name of a file (**tmp?**) that contains the following information:

   - the name of the file to be executed
   - the name of the input file (file returned from IBM)
   - the name of the IBM job
   - the programmer name
   - the IBM job number
   - the user's name from the "usr=" line
   - the user's login directory
   - the minimum file system space

2. *Shqer* uses two parameters. The first is the delay time between **log** file reads. The second is a *nice*(2) factor which is applied to any programs spawned by *shqer*. These values are defined in **/usr/include/rje.h** (**QDELAY** and **QNICE**).

3. When each log entry is read, the appropriate program is spawned with the following characteristics:
   - The returned RJE file is the standard input to the program.
   - The standard and diagnostic outputs are **/dev/null**.
   - The LOGNAME, HOME, and TZ variables are set to the appropriate values.
   - The arguments to the spawned program, in order, are:
     a. a numerical value indicating that the file system free space is equal or above (0) or below (1) **space** blocks (see Section 2.3).
     b. the IBM job name.
     c. the programmer name.
     d. the IBM job number.
     e. the user's login name.
4. After executing each program, the **tmp?** file and the returned RJE file are removed.

5. UTILITY PROGRAMS

5.1 Snoop

Snoop is the generic name of a program that can be used to trace the state of a VPM device and its associated communications line. *Snoop* depends on the *trace*(4) driver for its information. It reads trace entries from **/dev/trace** and converts them into a readable form that is printed on the standard output.

The usable name of *snoop* for a particular RJE subsystem is *snoopN*, where *N* is the low order three bits from the VPM minor device number. If VPM device names adhere to the **vpm0**, **vpm1**, ... **vpm***n* naming convention, each *snoop* name corresponds to its VPM device. In our hypothetical system, **vpm0** is used by the **rje1** subsystem, and **vpm1** is used by the **rje2** subsystem (see Section 2.3). Therefore, **/usr/rje1/snoop0** and **/usr/rje2/snoop1** are linked to **/usr/rje/snoop**.

Each *snoop* prints trace entries for its associated VPM device. Trace entries are printed in the following form:

sequence type information

where:

- **sequence** specifies the order of trace occurrences. It is a value between 0 and 99.
- **type** specifies the action being traced (e.g., transfers, driver activity).
- **information** describes data being transferred and driver activity.

The following table explains the meaning of trace **types** and their associated **information**.

type	information	meaning
CL	Closed	The VPM device has been closed.
CL	Clean	The VPM driver is cleaning up for this device.
OP	Opened	The VPM has been successfully opened.
OP	Failed(open)	The open failed because the device was already open.

OP	Failed(dev)	The open failed because the device number was out of range.
OP	Failed(set)	The open failed because the KMC could not be reset.
RR	Buf	The VPM script has returned a receive buffer to the VPM driver.
RX	Buf	The VPM script has returned a transmit buffer to the VPM driver.
RD	*num* bytes	*Num* bytes were read from the VPM device by *rjerecv*.
SC	Exit(*num*)	The VPM script has terminated. The VPM exit code is *num*. Exit codes are defined in *vpm*(4).
ST	Startup	The KMC has been started.
ST	Stopped	The VPM script has been stopped.
TR	Started	The script has started tracing.
TR	R-ACK	A two byte acknowledgement (ACK) string has been received from the remote system. This indicates that the previous transmission was properly received.
TR	S-ACK	A two byte acknowledgement (ACK) string has been transmitted to the remote system.
TR	R-NAK	A "not-acknowledged" (NAK) character has been received from the remote system. This indicates that the previous transmission was not properly received.
TR	S-NAK	A "not-acknowledged" (NAK) character has been transmitted to the remote system.
TR	R-ENQ	A enquiry (ENQ) character has been received from the remote system.
TR	S-ENQ	A enquiry (ENQ) character has been transmitted to the remote system.
TR	R-WAIT	The remote machine has requested that no data be transmitted to it.
TR	R-OKBLK	A valid data block was received from the remote machine.
TR	R-ERRBLK	An invalid Cyclic Redundancy Check (CRC) was received with a data block.
TR	R-SEQERR	The block sequence count on a received data block was invalid.
TR	R-JUNK	An invalid data block was received from the remote system.
TR	TIMEOUT	The remote machine did not respond within 3 seconds.
TR	S-BLK	A data block has been transmitted to the remote system.
WR	*num* bytes	*Num* bytes were written to the VPM device by *rjexmit*.

Trace entries of type **TR** are traces from the VPM script. Section 7.5 describes required responses to events and shows examples of typical *snoop* output.

5.2 Rjestat

This program is supplied as a user command. The program's two functions are to describe the status of the RJE subsystems and to provide a remote IBM status console. The remainder of this section describes these two functions.

5.2.1 RJE Status

When invoked, *rjestat* reports the status of the RJE subsystems. If remote system (**host**) names are specified, only those statuses are reported. *Rjestat* uses the following rules to report the status of a subsystem:

- *Rjestat* prints the contents of the file **status** if it exists in the subsystem directory. This file can contain any message the administrator wishes to have printed when users use *rjestat*.
- If the file **dead** exists in the subsystem's directory, the subsystem is not operating and the reason is contained in the file. *Rjestat* reports that RJE to **host** is down and prints the contents of the **dead** file as the reason.
- If the file **stop** exists in the subsystems directory, the *rjehalt* program has been used to inhibit that RJE subsystem. *Rjestat* reports that RJE to **host** has been stopped by the operator.
- If neither the **dead** nor the **stop** file exists, *rjestat* reports that RJE to **host** is operating normally.

Rjestat is supplied as the user's vehicle for checking the status of RJE. It is not meant to be an administrative tool; however, the reason for failure can be used to track the problem.

5.2.2 Status Console

To use *rjestat* as a status console, the *−shost* argument is used. *Rjestat* prints the status of the subsystem, then prompts with **host:** if the subsystem is up. Each console request is submitted to the RJE processes for transmission, and output is handled as specified. *Rjestat* checks the status prior to submitting each request, and will tell the user to try later if the subsystem goes down. *Rjestat* allows the **rje** or super-user logins to submit other than display requests. For a complete description of how to use the status console features, see *rjestat*(1C).

5.3 Cvt

This program converts any subsystem's **joblog** file to readable form. The first line printed is the process group number of the subsystem processes. The remaining output consists of entries in the following form:

file user-id records level

Where *file* is the name of the submitted file, *user-id* is the submitters user number, *records* is the number of "card" images, and *level* is the message level. The *records* and *level* fields are not used if the file name is **co*** (console request submitted by *rjestat*).

6. RJE ACCOUNTING

Each RJE subsystem will store accounting information in the **acctlog** file, if it exists. It is the responsibility of the RJE administrator to create and maintain this file in the subsystem's directory. Entries in this file describe RJE line use and are of the following form:

day time file user records

Each field is delimited by a tab character. The meanings of each field is as follows:

1. day — The day of occurrence in the form *mm/dd*.
2. time — The time of occurrence in the form *hh:mm:ss*.
3. file — The name of the UNIX file. The first two characters identify its type as follows:
   - **rd/sq** — the file was transmitted to the remote system
   - **pr** — the print output file was received from the remote system
   - **pu** — the punch output file was received from the remote system
4. user — The user ID of the user responsible for the transfer.
5. records — The number of records (card images) transferred for this file.

Because **acctlog** data is not used by RJE, it should not be allowed to grow too large. This can be accomplished by moving or processing the file during a system reboot (i.e., in **/etc/rc** *before* the RJE subsystems are started).

The following list describes some of the reports that could be generated from the **acctlog** data. Implementation of a program to produce accounting reports is the responsibility of the administrator.

- **Periodic Reports** — by using the **day** and **time** fields in the data, periodic usage reports can be produced.
- **By User Reports** — by using the **user** field in the data, usage-by-user reports can be produced.
- **By Subsystem Reports** — by using the **/usr/rje/lines** file information and each **acctlog** file, a usage-by-subsystem (or remote system) report can be produced.

Other reports can be produced using the type of file, size of jobs, etc.

7. TROUBLE SHOOTING

This section deals with RJE problems, and some methods for resolving them. The topics discussed in this section are as follows:

- Automatic Error Recovery
- Manual Error Recovery
- RJE Problems
- KMC/VPM Problems
- Trace Interpretation

7.1 Automatic Error Recovery

RJE attempts to be self-sustaining with respect to its availability. In general, if problems occur on the communications line or the remote machine (e.g., a crash) RJE will continually try to restart itself (this action will be referred to as a "reboot"). For example, if an RJE subsystem is started using *rjeload*, but the IBM system is not available, a fatal error will occur. The process that detects this error (usually *rjexmit* or *rjerecv*) will reboot the subsystem by executing *rjeinit* with a + as its argument. When *rjeinit* detects a + argument, it waits one minute before attempting to bring up the subsystem.

The *rjehalt* program can be used to prevent an RJE subsystem from rebooting itself when the remote system is not available for a known period of time. When the remote system is made available, the subsystem may be started in the normal way.

7.2 Manual Error Recovery

In order to manually recover from errors, one must know how to start and stop an RJE subsystem. There are two ways to start an RJE subsystem:

- *rje?load* — this program loads and starts the VPM script, and executes *rje?init*.
- *rje?init* — this program starts the *rje?* subsystem. In order to use this program, the VPM script must be loaded and started.

To stop the *rje?* subsystem, the *rje?halt* program should be executed. This stops the subsystem gracefully and will prevent a reboot.

The *rjeload* program must be used to start RJE for the first time (after a UNIX system reboot). Subsequently, as long as the script is running, execution sequences of *rjehalt* and *rjeinit* will stop and start RJE.

Manually starting and stopping RJE can be useful in tracking down problems. For example, if user jobs are not being submitted to the host machine, the following sequence can ease identification of the problem:

1. Halt the ailing subsystem.
2. Start a *snoop* process in the background with its output redirected to a file.
3. Restart the subsystem.
4. Scan the *snoop* output to determine where the problem is.

The *snoop* program is the most useful software tool for identifying RJE problems. Its uses are described in Section 7.5.

7.3 RJE Problems

This section describes problems that can occur in an RJE subsystem. These problems generally occur when the subsystem has not been set up properly. The following is a list of things to check to ensure that an RJE subsystem has been set up properly:

1. IBM description — the description of the remote UNIX machine must be consistent with the description in Section 2.2.
2. UNIX description — the file **/usr/rje/lines** must be set up properly (see Section 2.).
3. KMC/VPM setup — the VPM software must be installed and the proper VPM and KMC devices made. Each VPM device must correspond to the proper KMC device; see *vpm*(4).
4. Free space — as a general rule, all file systems must have a reasonable amount of free space. File systems containing RJE subsystems must have sufficient free space as described in Section 2.3 to ensure proper RJE operation.
5. Directories — each subsystem's directory and the controlling directory should be checked for the following:
   - All needed files exist.
   - The proper prefix is on each applicable RJE program.
   - The link count is correct for files that are linked.
   - All file and directory modes are correct.

   A sample subsystem directory and the controlling directory are shown in Section 3.
6. Initialization — peripherals information must be consistent on both systems (see Section 2.3). The line must be started on the IBM system, proper hardware connections made, etc.

Problems with a subsystem are indicated by error messages. *Rjeinit* checks for obstacles in bringing up RJE. If an obstacle is found, an error message indicating the obstacle is printed on the error output. If a problem is encountered during normal operation, the message is logged in the **errlog** file. This file, error messages, the output from *snoop*, and the checklist above should be used to determine and fix any subsystem problems. Generally, if a subsystem is set up properly but will not operate, the problem is the way the VPM or KMC has been set up, the remote system, or the hardware.

7.4 KMC/VPM Problems

This section describes the KMC and VPM uses, and problems that can occur. After installing KMC hardware and making KMC devices, all VPM software and devices must be made (see *vpm*(4)). The program *rjeload* links the devices to be used by the corresponding RJE subsystem.

The following is a list of items to check when problems occur:

1. Proper hardware — the line unit must be compatible with the modem and have the proper settings (see Section 2.1). Be sure that the KMC address and interrupt vector are correct.
2. Proper Devices — the major and minor device numbers for the KMC and VPM devices must be correct. It should also be verified that the *rjeload* program is called with the correct KMC and VPM device names.
3. Script runs — verify that the VPM script is able to run. This is done by tracing the proper VPM with the proper *snoop* program. *Snoop* will print "started" entries for both the KMC and VPM script (see Section 5.1). If no output appears from *snoop* when *rjeload* is executed, either the KMC is not working properly, or the KMC or VPM has not been set up properly (see items 1 and 2). Output of any other type from *snoop* should indicate where the problem is occurring.

7.5 Trace Interpretation

This section describes how to interpret trace output from the *snoop* program, and gives several examples. Section 5.1 describes the format and meaning of trace output lines, and should be read before this section.

Lines with type TR are traces from the VPM script. All others are driver traces and indicate the following:

- CL — activity occurring when the device has been closed.
- OP — activity occurring when the device has been opened.
- RD — read from device occurred.
- WR — write to device occurred.
- RR — a receive buffer has been returned.
- RX — a transmit buffer has been returned.
- ST — start or stop activity.
- SC — script exit type, exit value is given.

Section 5.1 enumerates all possible trace lines for each type, and describes the event. The remainder of this section consists of example trace output and its interpretation. Comments describing events will appear after the "*" in trace output. If more than one VPM were running, sequence numbers might not appear in order. For clarity, example sequences will be in order.

7.5.1 Normal RJE startup

The following is an example of trace output when RJE has been started up. In this case the remote machine responds to the enquiry byte (ENQ). The RJE subsystem signs on to the machine, then follows the handshaking protocol (exchanging ACKs).

```
Tracing vpm0
0   ST    Startup    * KMC started
1   TR    Started    * Script started
2   TR    S-ENQ      * Enquiry byte sent
3   ST    Start      * VPM Driver start
4   OP    Opened     * VPM Device open
5   TR    R-ACK      * Received acknowledgement
6   TR    S-ACK      * Handshaking
7   WR    84 bytes   * Signon record written
8   TR    R-ACK      * Handshaking
9   TR    S-BLK      * Sent signon block
10  TR    R-ACK      * Block acknowledged
11  RX    Buf        * Transmit buffer returned
12  TR    S-ACK      * Handshaking
13  TR    R-ACK      *    .
14  TR    S-ACK      *    .
15  TR    R-ACK      *    .
16  TR    S-ACK      *    .
17  TR    R-ACK      *    .
18  TR    S-ACK      *    .
19  TR    R-ACK      *    .
20  TR    S-ACK      * Handshaking
```

If any jobs had been submitted via the *send* command, or jobs were waiting to be returned, the traces would reflect the transfers rather than handshaking (see Section 7.5.3).

7.5.2 RJE startup—IBM not responding

This example shows trace output when RJE has been started, but does not receive a response from the remote machine. In general, the RJE script will timeout if a response is not received from the remote machine within 3 seconds of the last transmission. When a timeout is detected while starting up, the enquiry byte (ENQ) is retransmitted. This is repeated 6 times before the script gives up. Other timeout responses will be discussed later.

```
Tracing vpm0
86  ST    Startup    * KMC started
87  TR    Started    * Script started
88  TR    S-ENQ      * Enquiry byte sent
89  ST    Start      * VPM Driver start
90  OP    Opened     * VPM device open
91  WR    84 bytes   * Signon record written
92  TR    TIMEOUT    * No response to enquiry
93  TR    S-ENQ      * Enquiry byte sent
94  TR    TIMEOUT    * No response
95  TR    S-ENQ      * Enquiry byte sent
96  TR    TIMEOUT    * No response
97  TR    S-ENQ      * Enquiry byte sent
98  TR    TIMEOUT    * No response
99  TR    S-ENQ      * Enquiry byte sent
```

```
0   TR      TIMEOUT     * No response
1   TR      S-ENQ       * Enquiry byte sent
2   TR      TIMEOUT     * No response
3   RR      Buf         * Receive buffer returned
4   RD      1 bytes     * 1 byte read (error)
5   SC      Exit(0)     * Script exits normally
6   CL      Clean       * Cleanup done
7   ST      Stopped     * KMC stopped
8   CL      Closed      * VPM device closed
```

The above sequence will be repeated approximately every minute until a positive response is received from the host. During that minute the RJE subsystem is dormant, and the *rjestat* command will report that IBM is not responding. When this occurs, either the IBM machine is not available, down, line not started, etc., or there is a communications problem somewhere from where the KMC transmits data to where it receives data. The RJE administrator should first verify that the IBM machine is up, and the communications line has been started. If so, a hardware trace of the communications line should be done to aid in detecting the problem.

7.5.3 Transmitting and Receiving

This example shows trace output from the start of job transmission through its return. For simplicity, only one job is being transmitted and returned.

```
Tracing vpm0
94  TR      R-ACK       * Handshaking
95  TR      S-ACK       *    .
96  TR      R-ACK       *    .
97  TR      S-ACK       * Handshaking
98  WR      4 bytes     * Open reader request written
99  TR      R-ACK       * Handshaking
0   TR      S-BLK       * Sent open request block
1   TR      R-OKBLK     * Received block (grant)
2   RX      Buf         * Transmit buffer returned
3   RR      Buf         * Receive buffer returned
4   TR      S-ACK       * Block acknowledged
5   RD      7 bytes     * Read 7 bytes (grant)
6   TR      R-ACK       * Handshaking
7   TR      S-ACK       * Handshaking
8   WR      481 bytes   * First block written
9   WR      470 bytes   * Second block written
10  TR      R-ACK       * Handshaking
11  TR      S-BLK       * First block sent
12  TR      R-ACK       * Block acknowledged
13  RX      Buf         * Transmit buffer returned
14  WR      470 bytes   * Third block written
15  TR      S-BLK       * Second block sent
16  TR      R-OKBLK     * Received block (on reader msg)
17  RX      Buf         * Transmit buffer returned
18  RR      Buf         * Receive buffer returned
19  WR      470 bytes   * Fourth block written
20  RD      66 bytes    * Read 66 bytes (on reader msg)
21  TR      S-BLK       * Third block sent
22  TR      R-ACK       * Block acknowledged
23  RX      Buf         * Transmit buffer returned
24  WR      147 bytes   * Fifth block written
```

```
25  TR      S-BLK       * Fourth block sent
26  TR      R-ACK       * Block acknowledged
27  RX      Buf         * Transmit buffer returned
    :
    :
93  TR      R-ACK       * Handshaking
94  TR      S-ACK       * Handshaking
95  TR      R-OKBLK     * Received block (request)
96  RR      Buf         * Receive buffer returned
97  TR      S-ACK       * Block acknowledged
98  RD      7 bytes     * Read open printer request
99  TR      R-ACK       * Handshaking
0   TR      S-ACK       *    .
1   TR      R-ACK       *    .
2   TR      S-ACK       *    .
3   TR      R-ACK       *    .
4   TR      S-ACK       * Handshaking
5   WR      4 bytes     * Printer grant written
6   TR      R-ACK       * Handshaking
7   TR      S-BLK       * Block sent (grant)
8   TR      R-OKBLK     * First block received
9   RX      Buf         * Transmit buffer returned
10  RR      Buf         * Receive buffer returned
11  TR      S-ACK       * Block acknowledged
12  RD      64 bytes    * Read first block
13  TR      R-OKBLK     * Second block received
14  RR      Buf         * Receive buffer returned
15  TR      S-ACK       * Block acknowledged
16  RD      505 bytes   * Read second block
17  TR      R-OKBLK     * Third block received
18  RR      Buf         * Receive buffer returned
19  TR      S-ACK       * Block acknowledged
20  TR      R-OKBLK     * Fourth block received
21  RR      Buf         * Receive buffer returned
22  TR      S-ACK       * Block acknowledged
23  TR      R-ACK       * Handshaking
24  TR      S-ACK       *    .
25  TR      R-ACK       *    .
26  TR      S-ACK       * Handshaking
27  RD      470 bytes   * Read third block
28  RD      494 bytes   * Read fourth block
29  TR      R-ACK       * Handshaking
30  TR      S-ACK       * Handshaking
    :
    :
```

Requests and grants are part of the multi-leaving protocol. Appendix B of *OS/VS MVS JES2 Logic* (SY24-6000-1) describes this protocol in detail. When jobs are being transmitted and received simultaneously, as in a busier RJE subsystem, much less handshaking is involved. Rather than acknowledging blocks with ACKs, the protocol allows a block to be returned (this implies acknowledgement of the received block). The following example shows trace output at a busy time:

```
Tracing vpm0
41   TR      R-OKBLK    * Received block
42   RX      Buf        *
43   RR      Buf        *
44   TR      S-BLK      * Sent block
45   WR      493 bytes  *
46   RD      496 bytes  *
47   TR      R-OKBLK    * Received block
48   RX      Buf        *
49   RR      Buf        *
50   RD      65 bytes   *
51   WR      4 bytes    *
52   TR      S-BLK      * Sent block
53   TR      R-OKBLK    * Received block
54   RX      Buf        *
55   RR      Buf        *
56   TR      S-BLK      * Sent block
57   WR      493 bytes  *
58   RD      7 bytes    *
59   TR      R-OKBLK    * Received block
60   RX      Buf        *
61   RR      Buf        *
62   WR      493 bytes  *
63   RD      496 bytes  *
64   TR      S-BLK      * Sent block
65   TR      R-OKBLK    * Received block
```

Notice that because there is work to be done on both sides, acknowledgements are implied.

7.5.4 Timeout Error Recovery

This example shows activity resulting from timeouts occurring during normal operation. These timeouts were caused because the remote JES3 system has performance problems, and occasionally does not respond in the required three seconds.

```
Tracing vpm1
27   TR      S-ACK      * Handshaking
28   TR      R-ACK      *    .
29   TR      S-ACK      *    .
30   TR      TIMEOUT    * No response
31   TR      S-NAK      * Not acknowledged
32   TR      TIMEOUT    * No response
33   TR      S-NAK      * Not acknowledged
34   TR      R-ACK      * Response
35   TR      S-ACK      * Handshaking
36   TR      R-ACK      *    .
      .
      .
54   TR      R-ACK      *    .
55   TR      S-ACK      * Handshaking
56   TR      TIMEOUT    * No response
57   TR      S-NAK      * Not acknowledged
58   TR      R-ACK      * Response
59   TR      S-ACK      * Handshaking
      .
      .
```

The response to these timeouts are NAKs (not acknowledged). RJE will respond this way up to six times before giving up and attempting a reboot. At this time *rjestat* would report that there are "Line Errors." NAK is a request to retransmit the previous response.

7.5.5 Communication Line Errors

This example shows trace output from an RJE subsystem that uses a dial-up connection. The phone line is noisy and is prone to dropping.

```
Tracing vpm1
63  TR      S-ACK       * Handshaking
64  TR      R-ACK       *    .
65  TR      S-ACK       *    .
66  TR      R-JUNK      * Noise on the line
67  TR      S-NAK       * Not acknowledged
68  TR      R-ACK       * Recovery
69  TR      S-ACK       *
70  TR      R-ACK       *
71  TR      S-ACK       *
72  TR      TIMEOUT     * Line has dropped
73  TR      S-NAK       * Attempting to recover
74  TR      TIMEOUT     *    .
75  TR      S-NAK       *    .
    .
    .
80  TR      TIMEOUT     *    .
81  TR      S-NAK       *    .
82  TR      TIMEOUT     *    .
83  TR      S-NAK       *    .
84  RR      Buf         * Receive buffer returned
85  RD      1 bytes     * 1 byte read (error)
86  SC      Exit(0)     * Script exits
87  CL      Clean       * Cleanup
88  ST      Stopped     * KMC Stopped
89  CL      Closed      * VPM device closed
```

The error read in the above sequence causes RJE to reboot and *rjestat* to report line errors. If this were to occur frequently, a different method of communication should be used.

7.5.6 Error Responses

As seen in the sections above, the response to most errors is to send a NAK. The only exception is when starting up (see Section 7.5.2). Whenever a NAK is received on either side, it indicates that the previous transmission was not properly received. This should be followed by retransmission of the previous data. Generally, NAKs should not occur frequently, and should be followed by recovery. If errors occur frequently or NAKs do not cause recovery, the line should be checked for problems.

On some IBM systems, (e.g., JES2), an I/O error is printed at the system console whenever a NAK is received. These I/O errors can also be helpful in detecting the problem; however, they will not be discussed here as they vary with the system. It is assumed that someone in IBM support can assist if needed.

Release 1.0 of the UNIX Virtual Protocol Machine

P. F. Long
C. Mee, III

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This memorandum describes the initial release of the Virtual Protocol Machine (VPM), a new UNIX† synchronous communication subsystem. The VPM is built around the KMC11, a small, high-speed microcomputer that connects to the UNIBUS of a PDP-11 or VAX-11/780. The VPM is a software construct for implementing link protocols on the KMC11 in a high-level language.

A compiler, *vpmc*, is provided to translate a high-level description of a protocol (protocol script) into the instruction set of the virtual machine. *Vpmc* supports C-like control-flow constructs, a modest subset of C-like statements and expressions, and a set of communication primitives that permit implementation of byte-oriented protocols such as BISYNC. (Primitives that support bit-oriented protocols such as HDLC have been defined and will be available in a later release of VPM.) An interpreter is provided that runs in the KMC11 and interprets the virtual machine instruction set. A UNIX driver, *vpm.c*, provides the interface between the user process's *open*, *close*, *read*, and *write* calls and the protocol script being executed by the interpreter. Besides providing the benefits of a high-level language implementation of protocols, the VPM approach permits portable protocol implementations.

The VPM software consists of five components:

1. *vpmc*: a UNIX compiler for the protocol description language.
2. VPM interpreter: the KMC11 program that controls the overall operation of the KMC11 and interprets the protocol script.
3. *vpm.c*: the UNIX driver that provides the interface to the VPM.
4. *vpmstart*: a UNIX command that copies a load module into the KMC11 and starts it.
5. *vpmtrace*: a UNIX command that prints an event trace for debugging while the protocol is running.

The procedures for installation and use of the VPM commands and the VPM driver are described; the pertinent manual entries are attached.

INTRODUCTION

The Virtual Protocol Machine (VPM) is a new UNIX synchronous communications subsystem built around the KMC11 microcomputer. The KMC11 is a small, high-speed, 8-bit microcomputer manufactured by DEC. It connects to the UNIBUS of a PDP-11 or VAX-11/780 and can become UNIBUS master, thus giving it direct memory-access capability (DMA), as well as the ability to control other UNIBUS devices. While other DEC communications devices provide direct-memory access, the KMC11 is the only one that is also fully programmable. Thus the KMC11 can provide most of the CPU power and some of the address space required to do data communications, thereby relieving the main CPU of these burdens. All is not roses, however:

† UNIX is a trademark of Bell Laboratories.

the KMC11 must be programmed in an unfamiliar and somewhat awkward assembly language. This, together with a requirement to provide several varieties of the BISYNC protocol and with a need to support, in the future, other link protocols such as HDLC, was the motivation for the development of the VPM.

The VPM is a software construct for implementing link protocols on the KMC11 using a high-level language. A compiler, *vpmc*, is provided to translate a high-level description of a protocol (*protocol script*) into the instruction set of the virtual machine. *Vpmc* uses a variant of Ratfor [1] as a front end to provide control-flow constructs such as *if-else*, *for*, *while*, *switch*, and *repeat-until*, as well as other benefits. *Vpmc* supports a modest subset of C-like statements and expressions, plus a set of communications primitives that permit succinct and easily-understood implementations of byte-oriented protocols such as BISYNC. These primitives allow the protocol scripts to reflect the essential structure of the protocol, while hiding details that arise from a particular hardware-software environment. (Primitives that support bit-oriented protocols such as HDLC have been defined and will be available in a later release of VPM.) An interpreter is provided that runs in the KMC11 and interprets the virtual machine instruction set. This program also controls the communications line and provides the interface to the UNIX host machine. The compiled protocol script is loaded with the interpreter into the KMC11. A UNIX driver, *vpm.c*, provides the interface between the user process's *open*, *close*, *read*, and *write* calls and the protocol script executed by the interpreter in the KMC11. (The UNIX *kmc* driver is used to implement this interface.) For a pictorial overview of VPM, see Figures 1 and 2.

Besides providing the benefits of a high-level language implementation of protocols, such as ease of programming and maintainability, the VPM approach permits portable protocol implementations. Portability can be achieved in several ways. First, because the interpreter and the compiled protocol script execute in the KMC11, they are the same regardless of the software running in the main CPU or, for that matter, regardless of the CPU itself. For example, the same interpreter and compiled protocol script can be used for UNIX/RT on a PDP-11 or for UNIX on a VAX-11. More general forms of portability are also possible. The instruction set of the virtual machine can be translated into almost any assembly language using one of the UNIX macro processors, such as *m4* [2]. This does *not* require that the assembler for the target machine have a macro expansion capability. (We may use this approach in the future to translate protocol scripts into KMC11 assembly language, thus gaining speed over the present virtual machine interpreter.) Another possibility for portability arises because Ratfor is used as a front-end; by limiting a protocol script to a statement and expression syntax acceptable to a Fortran compiler, the protocol is portable to machines that support Fortran in a suitable real-time environment. Finally, minor changes to a protocol script will yield a C implementation of the protocol. With any of these methods, the functions provided by the primitives (including the interfacing with communication devices and the execution environment) must be supplied by suitable library routines or system calls.

RELEASE 1.0

Release 1.0 of VPM is restricted to byte-oriented, half-duplex protocols such as BISYNC. A separate KMC11-B is required for each communications link. Each KMC11 running VPM must be equipped with a suitable DMC11 line unit. A DMC11-DA line unit is required for operation at speeds up to 19.2K bits/sec; a DMC11-FA or DMC11-MD is required for operation at speeds of 56K bits/sec. The modem control available on the DMC11-DA line unit permits both inward and outward dial-up communication.

---

[1] B. W. Kernighan, *RATFOR—A Preprocessor for a Rational Fortran*, Bell Laboratories.

[2] B. W. Kernighan, *The M4 Macro Processor*, Bell Laboratories.

This release of the VPM software is intended for use with UNIX Edition 1.1 or later. Operation with other versions of UNIX has not been tested. The VPM software consists of five components:

1. *vpmc*: UNIX compiler for the protocol description language.
2. VPM interpreter: the KMC11 program that controls the overall operation of the KMC11 and interprets the protocol script.
3. *vpm.c*: the UNIX driver that provides the interface to the VPM.
4. *vpmstart*: a UNIX command that copies a load module into the KMC11 and starts it.
5. *vpmtrace*: a UNIX command to print a debugging event trace.

Manual entries for *vpmc*(1C), *vpmstart*(1C), *vpmtrace*(1C), and *vpm*(4) are attached to this memorandum. A release tape containing the VPM software and manual entries is available from the authors. Installation procedures are described in the appendix to this memorandum.

Acknowledgements

The idea of using the KMC11 to interpret a protocol description was suggested by L. A. Wehr. He also offered useful suggestions and criticisms as the project implementation progressed.

APPENDIX

Hardware Installation and Switch Settings

The KMC11 microprocessor and DMC11 line unit must be installed in adjacent slots of a PDP-11 or VAX-11/780 backplane. The microprocessor and line unit are interconnected by a one-foot mylar cable. The line unit is connected to a suitable modem by a 25-foot modem cable. The device address and interrupt vector address switches on the KMC11 should be set for the selected addresses. All switches and jumpers on the DMC11 line unit should be in the normal configuration prescribed by the relevant DEC maintenance manual with one exception: the NO CRC switch (switch S2 in switch pack number 1) should be in the ON position. The purpose of this switch setting is to inhibit hardware CRC generation. Hardware CRC generation is not used with this release of the VPM software.

Installing the VPM Software on a UNIX System

In order to read the release tape, change to the directory into which the *vpm* software is to be read (say, *vpmdir*), then execute:

```
cpio −iBdv </dev/rmt0
```

The executable programs, shell procedures, manual entries, and examples of protocol scripts will be read into the current directory and the following six subdirectories will be created and loaded: *util*, *plsrc*, *ratsrc*, *bisyncb*, *drvsrc*, and *demo*. *Util* will contain some processors that may be needed: *awk*, *cpp*, *kas*, *kasb*, *kunb*, *kun*, and *m4*. (These processors are provided in case the versions on your system are not compatible with the release tape.) *Plsrc* will contain the source required to make *pl*, the main pass of *vpmc*. *Ratsrc* will contain the source required to make *vratfor*, a modified version of Ratfor used as a preprocessor for *pl*. *Bisyncb* will contain the VPM interpreter source for the the KMC11-B. *Drvsrc* will contain the source required to make the UNIX driver, *vpm.c*, and the command *vpmtrace*. *Demo* will contain demonstration programs and programs for checking the operations of the KMC11 and the VPM software.

Installation of the VPM Driver and Commands

To add the VPM driver to a UNIX Edition 1.1 system, do the following:

1. Add the following line to the file */etc/master*:

   ```
   vpm   0     36    6     vpm   0     0     15    1     5
   ```

2. Add the following two lines to the file */usr/src/uts/cf/cfigpa* (or its equivalent) for each VPM line to be added:

   ```
   vpm   0      0       0
   kmc11 vector address priority
   ```

   If the KMC11s that are to be used have already been configured, the lines immediately above relating to KMC11s should *not* be added. See *config*(1M), *master*(5), and *Setting up UNIX* for more information.

3. To make a UNIX system that includes the VPM driver, copy *vpmmkdrv*, found in *vpmdir*, to */usr/src/uts/cp* or its equivalent. Check the *defines* at the beginning of *vpmmkdrv* to verify that the directories used are appropriate for your system. Then execute:

   ```
   vpmmkdrv sysname dfile
   ```

   where *sysname* is the name to be given to the system and *dfile* is the file modified in step 2 above. *Dfile* must be a simple file name (not a full path name).

4. To install the VPM commands, check the *defines* at the beginning of the shell procedure *vpmmkcmds* to verify that the directories used are appropriate for your system. Then execute:

   vpmmkcmds

5. Use *mknod*(1M) to create a node for each VPM line and each KMC11:

   /etc/mknod /dev/vpm? c *major minor*

   where *major* and *minor* are both octal; *major* is determined by *vpm*'s position in the *cdevsw* table and *minor* defines the KMC11 and VPM as follows: the two most significant bits denote the KMC11 number (0-3) and the three least significant bits denote the VPM number. For example, if KMC11s 2 and 3 are to be used for VPM, then the minor device numbers should be 0200 and 0301, respectively.

Compiling Protocol Scripts

The manual entry for *vpmc*(1C) describes the protocol description language. See also the examples of protocol scripts included on the release tape: *demo.r*, *demo.c*, *hasp.r*, and *mod40.r*.

When checking a protocol script for syntax errors, the **−c** option may be used.

Syntax errors detected by *ratfor* are noted as follows:

*****F ratfor:syntax error, line *n*, file *filen*

The line number *n* is in file *filen*.

Syntax errors detected by *pl* are noted as follows:

***** pl: syntax error, input line *n*.

To examine this line, a temporary file must be created as follows:

vpmc −m −r filen >temp

The temporary file can then be inspected using *ed*. The line number *n* refers to this file.

When all syntax errors have been eliminated, a KMC11 load module can be created by omitting the **−c** or **−r** options on the *vpmc* command.

Testing Protocols

When a load module suitable for testing has been made using *vpmc*, *vpmstart* may be used to load the file into the KMC11 and to start the interpreter. To view and record the trace records simultaneously execute:

vpmtrace | tee eventfile

A high-speed CRT terminal is best if you wish to get an impression of what is happening in real time. When a user program opens the VPM device, interpretation of the protocol script begins. Script interpretation ends if the VPM device is closed. Various error conditions can also terminate the script; they are described in *vpm*(4).

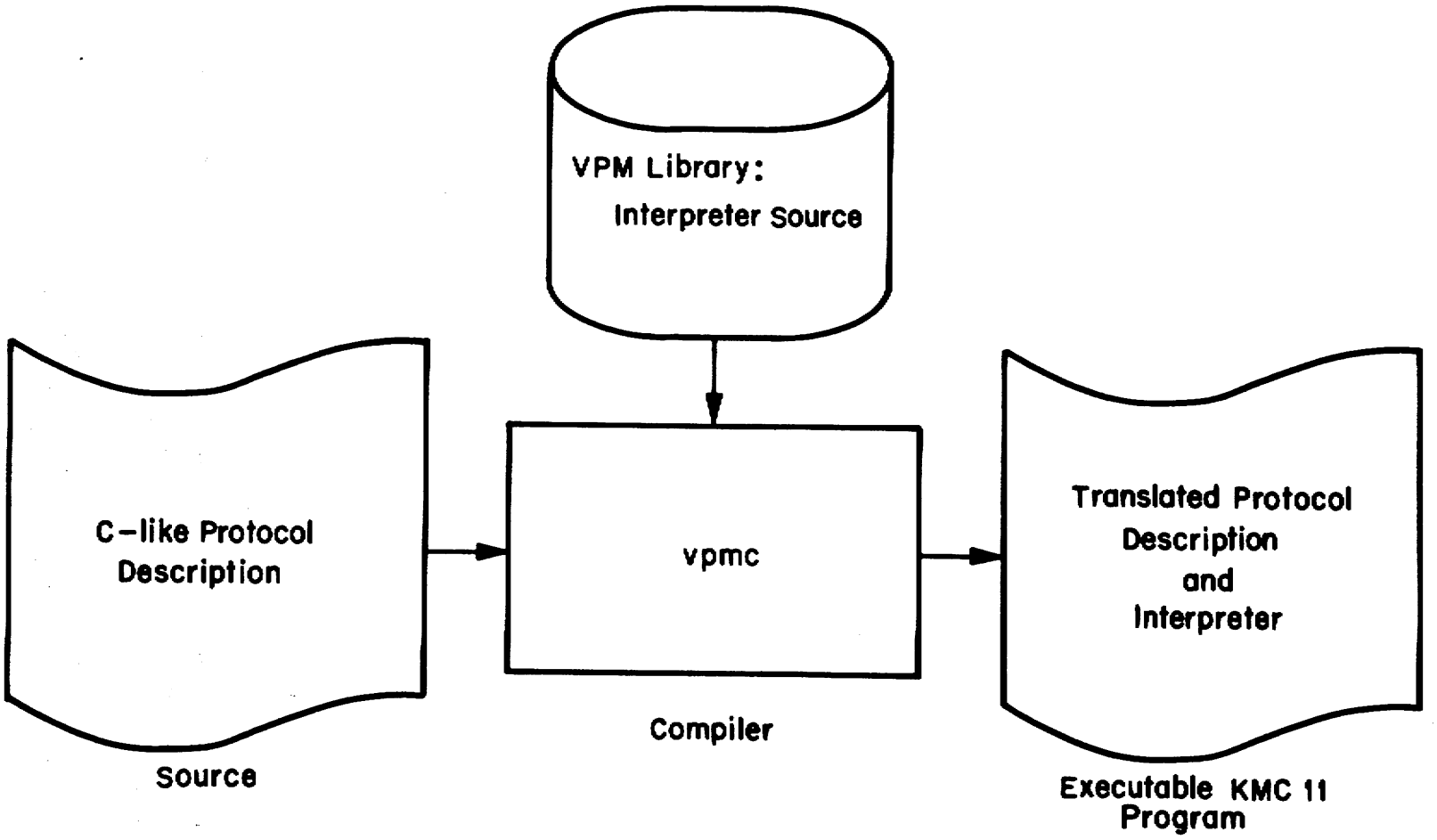
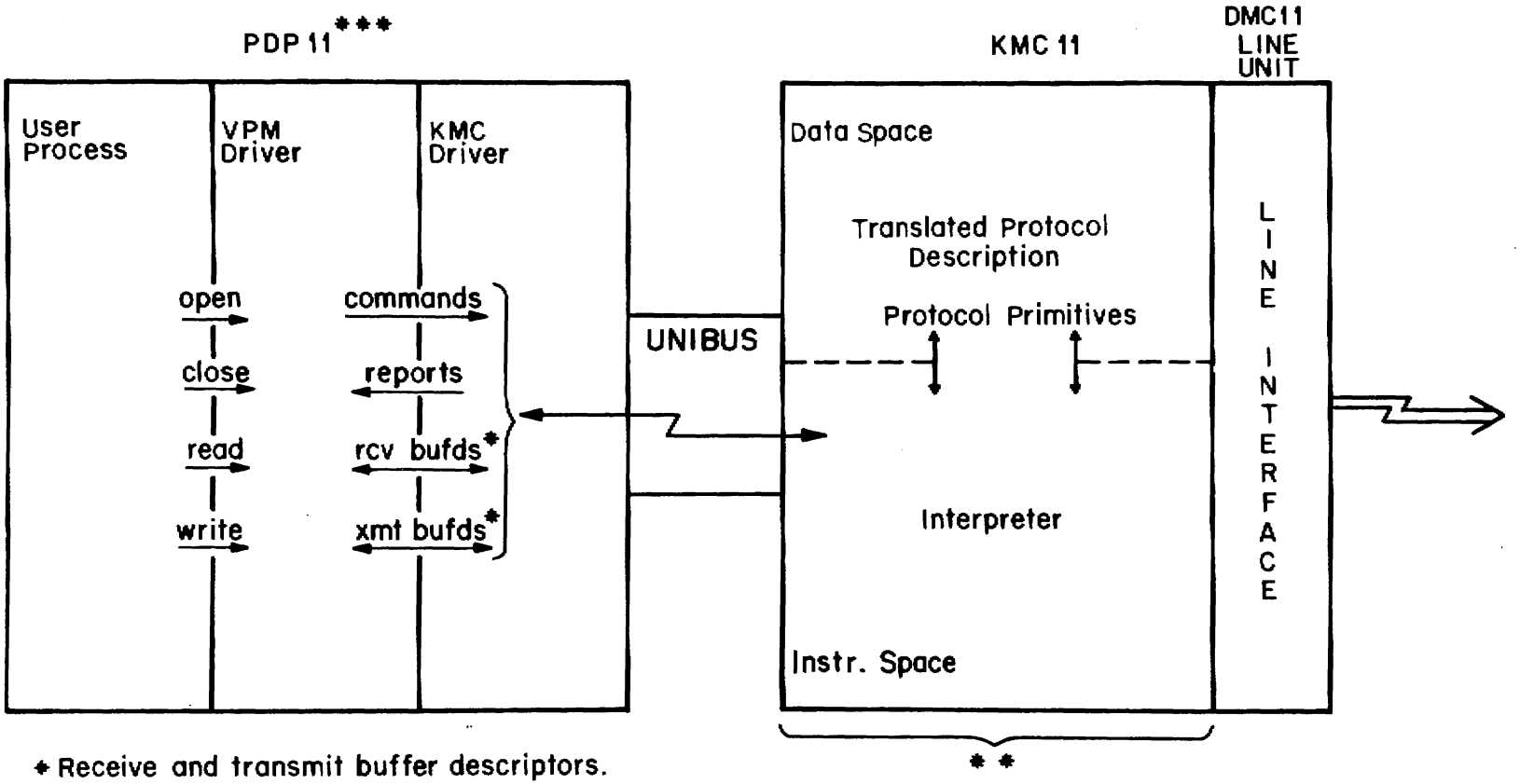


Figure 1
Protocol Compilation Process



* Receive and transmit buffer descriptors.

** Executable KMC11 program produced by *vpmc* & downloaded by *vpmstart*.

*** Release 2 will also run on the VAX.

Figure 2
VPM Components and Interfaces

Release 2.0 of the UNIX Virtual Protocol Machine

P. F. Long
C. Mee, III

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This memorandum describes the second release of the UNIX† Virtual Protocol Machine (VPM). VPM is a general-purpose synchronous UNIX communications interface that allows link-level protocols such as BISYNC and HDLC to be implemented on the KMC11-B (a DEC microcomputer) in a high-level language. The VPM software consists of a protocol compiler, a UNIX driver, an interpreter that executes in the KMC, and several utility programs.

The first release of VPM supports a class of byte-oriented half-duplex protocols collectively known as BISYNC. The present release adds support for bit-oriented, full-duplex protocols such as the international standard High-Level Data Link Control (HDLC). Other features of Release 2.0 include:

1. An increase in the number of buffers that the interpreter can accept at one time.
2. Additional debugging facilities.
3. Provisions for interprocess communication between the protocol script and a UNIX driver or a user process.
4. A cleaner separation of functions in the UNIX driver to facilitate tailoring of VPM to particular applications.

The procedures for adding VPM Release 2.0 to a UNIX 3.0 system and testing it to ensure proper operation are given.

Introduction

This memorandum describes the second release of the UNIX Virtual Protocol Machine (VPM). The first release was described in a previous memorandum [1], which should be read as background for this memorandum. See also the *UNIX User's Manual* [4] entry for *vpm*(4).

VPM is a general-purpose UNIX interface for synchronous communications lines. VPM allows link-level protocols such as BISYNC and HDLC to be implemented on the DEC KMC11-B microcomputer in a high-level language. The hardware required to support VPM is a PDP-11/70, /45, or /34, or a VAX-11/780 host computer, a KMC11-B microcomputer, and a DMC11-DA, -FA, or -FD synchronous communications interface. All of the above items are manufactured by DEC. The use of the KMC microcomputer allows the VPM to perform direct-memory-access (DMA) transfers to and from main memory. The link-level communications protocol is executed by the VPM interpreter running in the KMC microcomputer. This implementation technique leads to a portable protocol representation and efficient protocol execution.

The VPM software consists of a protocol compiler, a UNIX driver, an interpreter that executes in the KMC, and several utility programs. The compiler, which executes in the host computer, translates a protocol described in a high-level language into a load module for the KMC. The load module contains the VPM interpreter and a compiled representation of the protocol. The

† UNIX is a trademark of Bell Laboratories.

interpreter executes the protocol, communicates with the UNIX driver in the host computer, and controls the communications line interface.

The first release of VPM supported a large class of protocols collectively known as BISYNC. These protocols are distinguished by the use of control characters to provide framing and transparency. At the frame level, these protocols operate in a half-duplex manner, although they sometimes use full-duplex communications facilities to reduce the time required to reverse the direction of transmission.

Release 2.0 of VPM adds support for bit-oriented, full-duplex protocols. This class of protocols includes IBM's Synchronous Data Link Control (SDLC) and the international standard High-Level Data Link Control (HDLC). LAPB, a subset of HDLC which is the link-level protocol specified in the BX.25 Bell System Standard, has been implemented using VPM and is available with this release [2,3]. The interpreter used for bit-oriented protocols is different from that used for character-oriented (BISYNC) protocols. The appropriate interpreter is selected by means of a compiler option.

Other features of Release 2.0 include:

1. An increase in the number of transmit and receive buffers that the interpreter can accept at one time.
2. Additional debugging facilities.
3. provisions for interprocess communication between the protocol script and a UNIX driver or a user process.
4. A cleaner separation of functions in the UNIX driver to facilitate tailoring of VPM to particular applications.

Support for Bit-Oriented Protocols

The capability to use bit-oriented protocols such as HDLC is provided by a new set of communications primitives. These primitives are frame-oriented and non-blocking, whereas the BISYNC primitives are character-oriented and blocking. The new primitives are fully described in the manual entry for *vpmc*(1C). An overview of these primitives follows.

The VPM interpreter maintains a set of queues for transmit buffers and another set of queues for receive buffers. When a transmit buffer is passed to the KMC by the UNIX driver, the buffer is appended to the unopened-transmit-buffer queue. The protocol script in the KMC obtains a transmit buffer from the unopened-transmit-buffer queue by means of the *getxfrm* primitive; the buffer is then said to be *open*. In order to get (open) a transmit buffer, the script must provide a transmit-sequence number. This sequence number must be in the range 0-7 and must be distinct from the sequence number currently assigned to every other currently-open transmit buffer. This sequence number is used to identify the buffer for subsequent calls to the *xmtfrm* and *rtnxfrm* primitives. The *xmtfrm* primitive initiates transmission of the specified buffer, using the control information specified by a previous *setctl* primitive. Transmission proceeds asynchronously. The script can test for completion of an output transfer by means of the *xmtbusy* primitive. Open transmit buffers can be transmitted any number of times. When the script decides that a buffer has successfully been received at the destination, it notifies the interpreter by means of the *rtnxfrm* primitive. This causes the buffer to be placed on the transmit-buffer-return queue; the buffer is then no longer considered to be open and the sequence number can be reused. The driver is notified as soon as possible that the buffer has been closed. The buffer is then removed from the transmit-buffer-return queue.

When a receive buffer is passed to the KMC by the driver, the buffer is placed on the empty-receive-buffer queue. When the first byte of a new frame arrives, an empty receive buffer is obtained from the empty-receive-buffer queue and the incoming characters are placed into the buffer as they arrive. An incoming frame will be discarded if the frame is too short (less than four bytes including CRC), if the frame is too long to fit in the receive buffer, or if the CRC is incorrect. If a frame is received successfully, the buffer is placed on the completed-receive-

frame queue, otherwise the buffer is returned to the empty-receive-buffer queue. When the script executes a *rcvfrm* primitive, the buffer at the head of the completed-receive-frame queue is removed from that queue and becomes the current receive buffer. If the script subsequently executes a *rtnrfrm* primitive before executing another *rcvfrm* primitive, the current receive buffer is placed on the receive-buffer-return queue. If the script executes a *rcvfrm* primitive before executing a *rtnrfrm* primitive, the current receive buffer, if any, is returned to the empty-receive-frame queue. Buffers on the receive-buffer-return queue are returned to the driver at the first opportunity.

If the empty-receive-buffer queue is empty when the first byte of a new frame is received, the first five bytes of the frame are retained in a staging area and the remainder of the frame is discarded. This allows a protocol script to receive a control frame (up to seven bytes including CRC) when no data buffer is available. When the next *rcvfrm* primitive is executed, the script will receive the information in the staging area along with an indication that the remainder of the frame has been discarded. If another frame arrives while the staging area is thus occupied, the new frame is discarded entirely.

A count is kept of the number of frames discarded for each reason. These counters may be read and reset from the host computer.

The VPM Split Driver

Because the VPM interpreter and a protocol script generally use most of the memory of the KMC, any higher levels of protocol that are required must be executed by the host CPU. The purpose of the VPM split driver is to provide a framework in which higher-level protocols can be implemented conveniently using low-level routines in the VPM driver to communicate with the interpreter in the KMC.

A set of functions has been written that provides a general-purpose interface to the link-level protocol being executed by the interpreter in the KMC. Their capabilities include a means to queue transmit and empty receive buffers for use by the protocol script in the KMC, to start and stop the script, and to send commands to and receive reports from the script. A means of getting a copy of and resetting the VPM interpreter's error counters is also provided. These functions will be referred to as interface functions or collectively as the interface module. Appendix 1 contains a description of each of these routines.

To implement higher levels of a protocol as a UNIX device driver, a set of routines must be written to implement the standard UNIX system calls: *open, close, read, write,* and *ioctl* as well as the required protocol. These routines will be referred to as protocol functions or collectively as a protocol module. The standard VPM driver does not implement a higher-level protocol but instead provides a transparent user interface that can be used by applications that supply their own higher levels of protocol. This driver can be used as an example for those interested in writing a different protocol module. Appendix 2 contains a description of these routines.

At least two other protocol modules have been written thus far. They are the Synchronous Terminal Interface (see *st*(4)) and the BANCS THP Interface.

Release 2.0 of VPM allows up to four different VPM protocol modules to be executing simultaneously. One KMC and one interface-module minor device[1] are required for each protocol being executed. Any number of protocol modules may be implemented, but no more than four can be in use at any one time because no more than four KMCs are supported. In general, each

1. Strictly speaking, the interface module is not a driver and therefore does not have minor devices; however, the minor device number in this case selects an element of the data-structure array associated with the interface module in the same way that the minor device number associated with a driver selects an element of a data-structure array.

protocol module can have up to 256 minor devices. The VPM Release 2.0 protocol module, however, can have at most 16 minor devices; this restriction is due to the fact that the minor device number of the VPM protocol module is used not only to specify the VPM minor device but also to specify the interface-module minor device and the KMC minor device. The low-order four bits of the protocol-module minor device number determine the protocol-module minor device; the next two bits determine the interface-module minor device; the next two bits determine the KMC minor device.

Transmit buffers and receive buffers are passed between the VPM interpreter, the interface module, and the protocol module by means of pointers to data structures known as *buffer descriptors*. The buffer-descriptor structure is defined as follows:

```
struct vpmbd {
        short   c_ct;           /* Buffer size */
        short   d_adres;        /* Low-order 16 bits of buffer address */
        char    d_hbits;        /* High-order 2 bits of buffer address */
        char    d_type;         /* Protocol-dependent */
        char    d_sta;          /* Protocol-dependent */
        char    d_dev;          /* Protocol-dependent */
        struct buf *d_buf;      /* Pointer to system buffer descriptor */
        int     d_bos;          /* Index of next byte in buffer */
        int     d_vpmtdev;      /* Minor device number */
}
```

For empty receive buffers, *c_ct* must be equal to the buffer size in bytes; for transmit buffers, *c_ct* must be equal to the number of bytes to be transmitted. When a receive buffer is returned to the protocol module, *c_ct* is equal to the number of data bytes in the buffer. *D_adres* and *d_hbits* must contain an 18-bit UNIBUS-mapped buffer address; the low-order 16 bits must be in *d_adres* and the high-order two bits must be in the low-order two bits of *d_hbits*. *D_type*, *d_sta*, and *d_dev* are protocol-dependent; when using the BISYNC interpreter these three bytes may be read and modified by the protocol script. See the discussion of *getxbuf*, *getrbuf*, *rtnxbuf*, and *rtnrbuf* in the manual entry for *vpmc*(1C). *D_buf* contains a pointer to a system buffer descriptor; this is used to return the buffer to the system buffer pool. *D_bos* is the index of the first byte in the buffer not yet returned to the user. *D_vpmdev* is the minor device number of the protocol-module minor device to which the buffer is allocated.

The Trace Driver

The trace driver provides a means by which a user program can receive trace information generated by the VPM driver and the protocol script to aid in debugging new protocol modules and protocol scripts. It may also be used to debug other drivers or system code not related to the VPM driver. This driver can be configured to have a number of minor devices. Each minor device provides a means by which a user program can read data generated by functions within the operating system. This data is recorded by calls to *trsave* as described in Appendix 3. Each call to *trsave* generates a unit of data known as an *event record* which consists of a channel number (one byte), a count (one byte) and *count* bytes of data. The channel number can be used to multiplex up to 16 data streams on each minor device.

Associated with each minor device of the trace driver is a *clist* queue which is used to save event records provided a user program has that minor device open and has enabled the channel to which the event records were written. Channels may be enabled in any combination, using the *ioctl* command VPMTRCO. See the manual entry for *trace*(4). While a minor device read queue is full, event records for that minor device are discarded. Appendix 3 contains a description of each trace-driver routine.

Minor device 0 of the trace driver is used by the VPM driver to record a variety of debugging information generated within the VPM driver and also to record the data generated by the *trace*

primitive in a protocol script. Minor device 1 of the trace driver is used to record the information generated by the *snap* primitive in a protocol script. The *vpmtrace* and *vpmsnap* commands are available for reading and formatting the data passed via these two minor devices. These two commands are described in the manual entry for *vpmstart*(1C). Appendix 4 contains a description of the VPM driver event trace.

Miscellaneous Improvements

Two new primitives have been added to the protocol language to allow communication between the link-level protocol script in the KMC and a higher-level protocol implemented in a user program or a VPM protocol module. The *getcmd* primitive allows the script to receive a four-byte command from a user program or a protocol module. The standard VPM protocol module allows a user program to pass a command to the script via an *ioctl* system call. Other VPM protocol modules can pass a command to the script by calling the *vpmcmd* routine in the VPM interface module. The *rtnrpt* primitive allows the script in the KMC to send a four-byte report to a protocol module or to a user program. The standard VPM protocol module allows a user program to receive a script report by means of an *ioctl* system call. A protocol module can receive reports from the interface module by calling the *vpmrpt* routine of the VPM interface module.

The *trace* primitive of the protocol language has been augmented to allow two arguments. The form with one argument is still supported; if only one argument is given, the second argument is assumed to be zero. A *snap* primitive has been added. This primitive causes four bytes of data from the script followed by a four-byte time stamp to be placed on the read queue for trace driver minor device 1.

The *timeout* primitive provided in Release 1.0 has been supplemented by a new *timer* primitive that allows a script to initialize a timer or test its current value. If the argument to *timer* is non-zero, the timer is initialized with the value of the argument. The timer is decremented ten times a second until the timer reaches zero. If the *timer* primitive is called with an argument of zero, it returns the current value of the timer. This value is zero if the timer has expired, otherwise non-zero.

In release 1.0 of VPM, the interpreter would accept at most one transmit buffer and one receive buffer at any given time. In Release 2.0 the interpreter will accept up to four transmit buffers and four receive buffers at a time. This applies to both the character-oriented (BISYNC) interpreter and the bit-oriented (HDLC) interpreter.

For applications with requirements for monitoring the integrity of the computer hardware and software, a form of cross-checking between the UNIX driver and the KMC has been implemented. Every three seconds the VPM interpreter in the KMC sends an "I'm-OK" report to the host; the host responds by sending an "I'm-OK" command to the KMC. If either the host or the KMC does not receive the "I'm-OK" signal within a reasonable time period, an error termination occurs.

Appendix 5 contains detailed instructions for adding VPM Release 2.0 to a UNIX 3.0 system. Appendix 6 describes a number of test programs and procedures that may be used to check the VPM hardware and software and to gain familiarity with the system.

Acknowledgements

We would like to thank our supervision, especially R. C. Haight and G. W. R. Luderer, for their support of the Virtual Protocol Machine. L. A. Wehr provided the initial idea of interpreting protocol descriptions with the KMC and helped us with debugging and useful advice from time to time. R. V. Baron of Department 9362 suggested a number of new features that became part of this release. R. M. Ermann of Department 5251 wrote the protocol script for the LAPB protocol and suggested several improvements in the HDLC primitives.

References

[1] Long, P. F. and Mee, C., III. *Release 1.0 of the UNIX Virtual Protocol Machine,* Bell Laboratories.

[2] Ermann, R. M. *Formal Specification of X.25 Compatible Link Protocol,* Bell Laboratories.

[3] Ermann, R. M. *Portable Implementation of BX.25 Level 2,* Bell Laboratories.

[4] Dolotta, T. A., Olsson, S. B., and Petruccelli, A. G. (eds.). *UNIX User's Manual*—Release 3.0, Bell Laboratories.

Appendix 1: The VPM Interface Module

The VPM interface functions provide a general-purpose interface between a higher-level protocol implemented in a VPM protocol module and the link-level protocol script executed by the VPM interpreter in the KMC. The KMC driver is used by the interface functions to pass commands to and receive reports from the VPM interpreter. When reports are received by the interface module that must be passed on to the protocol module, the protocol module's receive-interrupt routine (*vpmtrint* in the case of the standard VPM protocol module) is called.

This appendix describes each interface function. *Dev* is an argument to many of the interface functions and has the same meaning for all but two of them: the low-order four bits of the *dev* argument are *not* used by the interface functions; the next two bits determine the interface module minor device number; the next two bits determine the KMC minor device. Although *dev* is declared as an *int*, only the low-order eight bits are meaningful at this time. In calls to the *vpmtrace* and *vpmsnap* routines, *dev* need not be a minor device number because it is just saved as part of the event record. The definition of *dev* will not be repeated for each function.

vpmcmd (dev, cmd)
int dev;
char *cmd;

This function passes a command to the script. *Cmd* is the address of a four-byte array. The four bytes are passed to the VPM interpreter, which saves them until the protocol script executes a *getcmd* primitive. Only the most recent four bytes passed by a *vpmcmd* call are saved by the VPM interpreter.

struct vpmbd *vpmdeq (clp)
struct clist *clp;

This function removes the buffer-descriptor pointer at the head of the queue pointed to by *clp* and returns it to the caller. If the queue is empty, a null pointer is returned.

vpmemptq (dev, bdp)
int dev;
struct vpmbd *bdp;

This function is used to pass an empty receive buffer for use by the interpreter in the KMC. *Bdp* is a pointer to a buffer descriptor or null. If *bdp* is not a null pointer, the buffer descriptor is appended to the empty-receive-buffer queue for the interface module specified by *dev*. If the VPM interpreter currently has room for another empty receive buffer, the buffer at the head of the queue is removed and passed to the KMC. The sum of the number of buffers on the empty-receive-buffer queue and the number of receive buffers the VPM interpreter has in its queues is returned to the caller. If *bdp* is a null pointer, the above sum is returned and nothing else is done.

vpmxmtq (dev, bdp)
int dev;
struct vpmbd *bdp;

This function is used to pass a transmit buffer to the interpreter in the KMC. *Bdp* is a pointer to a buffer descriptor or null. If *bdp* is not a null pointer, the buffer descriptor is appended to the transmit-buffer queue for the interface module specified by *dev*. If the VPM interpreter currently has room for another transmit buffer, the buffer at the head of the queue is removed and passed to the KMC. The sum of the number of buffers on the transmit-buffer queue and the number of transmit buffers the VPM interpreter has in its queues is returned to the caller. If *bdp* is a null pointer, the above sum is returned and nothing else is done.

vpmenq (bdp, clp)
struct vpmbd *bdp;
struct clist *clp;

If *bdp* is a null pointer, the number of buffer-descriptor pointers on the *clist* queue pointed to by *clp* is returned. If *bdp* is not a null pointer, the buffer descriptor pointed to by *bdp* is appended to the *clist* queue pointed to by *clp* and the number of pointers currently on that queue is passed as the return value.

char *vpmerrs (dev, n)
int dev, n;

This function is used to read and reset the error counters in the VPM interpreter. The function passes a GETECMD command to the VPM interpreter and blocks until the interpreter responds; this command causes the interpreter to copy its error counters to an array in the interface module and send a completion report to the driver. After the copy operation is completed, a pointer to the error-count array is passed to the caller as the return value. The second argument is not currently used.

char *vpmrpt (dev)
int dev;

This function is used to receive a script report from the KMC. When the protocol script executes a *rtnrpt* primitive, four bytes of data are passed to the interface module. If a *rtnrpt* has been executed by the protocol script since the last call to *vpmrpt*, a pointer to the four bytes passed by the most recent *rtnrpt* primitive is returned; otherwise zero is returned.

vpmsave (type, dev, word1, word2)
char type, dev;
short word1, word2;

This function creates an event record with the following structure:

```
struct {
        short   c_seqn;         /* Sequence number */
        char    c_type;         /* Argument type */
        char    c_dev;          /* Argument dev */
        short   c_word1;        /* Argument word1 */
        short   c_word2;        /* Argument word2 */
}
```

This event record is passed to the *trace* driver using *trsave*.

vpmsnap (type, dev, word1, word2)
char type, dev;
short word1, word2;

This function is similar to *vpmsave*. The only difference is that a time stamp *(long s_lbolt)* is added to the event record after *word2*. A protocol script may generate a time-stamped event record by executing the *snap* primitive.

vpmstart (dev, type, rint)
int dev, type;
int (*rint)( );

This function must be called on the first open of the protocol-module minor device associated with the interface-module minor device and KMC identified by *dev*. *Type* is a number that identifies the program running in the KMC and must agree with the value specified when the KMC load module was loaded into the KMC. For VPM interpreters, *type* is conventionally 6. *Rint* is the name of a protocol-module routine to be called by the interface module when it needs to return a transmit buffer, a receive buffer, a script report, or an error-termination code. See the description of *vpmtrint* in Appendix 2 for an example of such a routine. *Vpmstart* sends a RUN command to the VPM interpreter which causes it to begin execution of the protocol script. If the interface module identified by *dev* is not configured, ENXIO is returned. If the module is already running, i.e., *vpmstart* has been called and *vpmstop* has not been called, or if the KMC is not running or was loaded using a different magic number, EACCES is returned. A return value of zero indicates a normal completion.

vpmstop (dev)
int dev;

This routine is called to halt the execution of the protocol script by the interpreter. The routine waits until the last transmit buffer has been returned by the protocol *script*, or until five seconds have elapsed, and then sends a HALT command to the VPM interpreter which causes the interpreter to stop executing the protocol script. When the interpreter acknowledges the HALT command, or after five seconds, any transmit or receive buffers still enqueued on the interface module's transmit- and empty-buffer queues are returned to the protocol module. This does not include buffers contained in the interpreter's queues. Generally, when the protocol script is halted normally, the interpreter will have one or more empty receive buffers. If the interpreter or protocol script terminates in error, some transmit buffers may also remain unaccounted for. The upshot of this is that a protocol module must keep a record of all buffers in use for each particular minor device, so that these buffers can be returned to the pool of available buffers when that minor device is closed.

Appendix 2: The VPM Protocol Module

This appendix gives a detailed description of the functions that make up the standard VPM protocol module. The description may be useful as a guide in writing other VPM protocol modules. The *dev* argument to the following routines is declared as an *int*; however, only the low-order eight bits are meaningful at this time. The low-order four bits are used to determine the minor device of the protocol module; the next two bits determine the minor device of the interface module; the next two bits determine the KMC minor device.

vpmopen (dev, flag)
int dev, flag;

This function opens the protocol-module minor device specified by the low-order four bits of *dev*. *Flag* contains the option bits specified on the *open* system call. Exclusive or non-exclusive *open*s are permitted. If the driver is opened for both reading-and-writing, the *open* is exclusive, i.e., no further *opens* are permitted. If the device is opened for reading only or for writing only, the *open* is non-exclusive and subsequent *open*s for reading only or writing only are permitted. If this device is not open when this function is called, it obtains a number of non-addressable system buffers to be used as receive buffers and passes them to the VPM interpreter using the interface routine *vpmemptq*. *Vpmopen* also calls the interface routine *vpmstart* if the minor device was not already open.

vpmclose (dev)
int dev;

This function closes the minor device specified by the low-order four bits of *dev*. It calls the interface routine *vpmstop*, flushes the receive queue for the specified minor device, releases its buffers, and reinitializes its data structure.

vpmwrite (dev)
int dev;

This function implements the *write* system call. If the transmit queue is not full, the function obtains a non-addressable system buffer, copies up to 512 bytes of the user's *write* data into it, and enqueues the buffer on the level 2 transmit queue using the interface function *vpmxmtq*. These steps are repeated until all of the user's *write* data has been copied. If the transmit queue is full when this function is called or if it becomes full while the function is executing, the calling process is blocked until there is room in the queue for more transmit buffers.

vpmread (dev)
int dev;

This function implements the *read* system call. When it is called, the calling process is blocked until the receive queue is non-empty. As data is received by the VPM interpreter, it is placed into an empty receive buffer. When the protocol script decides that the data contained in a particular buffer is valid, it executes a *rtnrbuf* (BISYNC) or *rtnrfrm* (HDLC) primitive which causes the buffer descriptor pointer to be passed to the interface modules interrupt routine. The interface module then passes the buffer descriptor pointer to the protocol module by calling the protocol module's interrupt routine. The protocol module enqueues the buffer descriptor pointer on the receive queue and wakes up (unblocks) the reader(s). The number of bytes requested, or the data in one buffer, whichever is less, is copied to the user process; the number of bytes copied is passed as the return value. Any bytes remaining in a buffer are used to satisfy subsequent *read* requests.

vpmioctl (dev, cmd, arg, mode)
int dev, cmd, mode;
char *arg;

This function implements the *ioctl* system call. *Cmd* determines the function to be performed as follows:

VPMCMD — Pass a command to the protocol script. The first four bytes of the array pointed to by *arg* are passed to the VPM interpreter which saves them and passes them to the protocol script the next time it executes a *getcmd* primitive.

VPMERRS — Get and reset the VPM interpreter's error counters. The eight-byte array containing the VPM interpreter's error counters is copied to the user array pointed to by *arg*. The interpreter's copy of the error counters is then set to zero.

VPMRPT — Get a report from the protocol script. If the protocol script has executed a *rtnrpt* primitive since the last time this *ioctl* command was issued, the script report (four bytes) is copied to the user array pointed to by *arg* and *one* is passed as the return value; otherwise, *zero* is passed as the returned value.

The *mode* argument is not used. The values for VPMCMD, VPMERRS, and VPMRPT are defined in file */usr/include/sys/vpm.h*.

vpmtrint (dev, code, bdp)
int dev, code;
struct vpmbd *bdp;

The address of this function is passed to the protocol module using the *vpmstart* function described in Appendix 1. This routine is called from the interface module to return transmit buffers, receive buffers, script reports, or error termination codes. It is usually called at interrupt priority and therefore must not sleep or do unnecessary work. *Code* identifies the purpose of the call and determines the meaning of *bdp* as follows:

RRTNXBUF — *Bdp* is a pointer to the buffer descriptor for a transmit buffer. This call is made when the protocol script executes a *rtnxbuf* (BISYNC) or a *rtnxfrm* (HDLC).

RRTNRBUF — *Bdp* is a pointer to the buffer descriptor for a receive buffer. This call is made when the protocol script executes a *rtnrbuf* (BISYNC) or a *rtnrfrm* (HDLC).

RRTNEBUF — *Bdp* is a pointer to the buffer descriptor for an empty receive buffer. This call is used to return empty receive buffers when the interface module is stopped by calling *vpmstop*.

ERRTERM — *Bdp* is the error-termination code passed to the interface module by the VPM interpreter when it halts the protocol script because of an error condition. The meaning of these error codes is given in the manual entry for *vpm*(4).

The values for RRTNXBUF, RRTNRBUF, RRTNEBUF, and ERRTERM are defined in file */usr/include/sys/vpm.h*.

Appendix 3: The Trace Driver

The trace driver provides a means by which a user program can receive trace information generated by the VPM driver, a protocol script, or some other driver. See the manual entry for *trace*(4).

A description of each routine of the trace driver follows.

tropen (dev)
int dev;

This function opens the minor device specified by *dev* exclusively.

trclose (dev)
int dev;

This function closes the minor device specified by *dev*. It discards any data on the read queue and initializes the data structure associated with the minor device.

trread (dev)
int dev;

This function implements the *read* system call; it sleeps until at least one event record is available on the read queue associated with *dev*. It then copies records to the user until the user's read count is less than the number of bytes in the next event record or until the read queue is empty. The number of bytes copied is passed as the return value.

trioctl (dev, cmd, arg, mode)
int dev, cmd, arg, mode;

This function implements the *ioctl* system call. *Cmd* indicates the operation to be performed. The driver has one command:

> VPMTRCO — Enable a trace channel. In order for data to be saved on the read queue for minor device *dev*, the device must be open and the channel to which it is written must be enabled. This command enables channel *arg*, which must be in the range 0 to 15. Any combination of channels may be enabled by repeatedly calling this function with different values of *arg*. All channels are disabled when the minor device is closed.

trsave (dev, chno, buf, ct)
char dev, chno, *buf, ct;

If minor device *dev* of the trace driver is open and channel *chno* of that minor device is currently enabled then *chno* and *ct*, followed by *ct* bytes starting at address *buf*, are copied onto the read queue associated with *dev*, provided the read queue for that device has room for the complete event record. If there is not room for the complete event record, the record is discarded.

Appendix 4: The VPM Event Trace

Calls to the interface routine *vpmsave* have been placed strategically throughout the standard VPM protocol module (*vpmt.c*) and the VPM interface module (*vpmb.c*) to provide an event trace for debugging new protocol modules and/or protocol scripts. A protocol script may generate an event record by executing a *trace* primitive. All such event records are discarded unless some user program has opened minor device 0 of the trace driver and enabled channel 0 of that minor device. The command *vpmtrace*(1C) opens this device and enables channel 0, then reads event records and prints them on the standard output as they are received. Each kind of event record that is generated by the VPM driver will be described by giving the *vpmsave* function call as it appears in *vpmt.c* or *vpmb.c*, followed by an example of the line printed by *vpmtrace* as a result of this call. Following this, the context of the *vpmsave* call and the definition of the parameters passed will be given. The definition of a parameter that appears in more than one call will not be repeated. The first five calls to *vpmsave* occur in the source file *vpmt.c*; the remaining calls occur in *vpmb.c*.

vpmsave ('p', dev, ec, 0)

243 p 100 15 0

Called if *vpmstart* returns an error code. The first field of the printed record contain a sequence number assigned by *vpmsave*. The remaining four fields contain the four remaining arguments to *vpmsave* in the same order as they appear in the call to *vpmsave*. The first argument to *vpmsave*, in this case a 'p', identifies the record type. *Dev* is the minor device number as defined earlier; *ec* is the value returned by *vpmstart*.

vpmsave ('o', dev, vp->vt_state, 0)

244 o 100 1 0

Called just before the normal return point of *vpmopen*. The variable, *vp->vt_state*, contains the state bits for the protocol module. Refer to the source file, *vpmt.c*, for the definitions of the state bits.

vpmsave ('c', dev, vp->vt_state, 0)

245 c 100 13 0

Called from *vpmclose* just before the state bits are initialized.

vpmsave ('w', dev, ct, dp)

246 w 100 1000

Called just before putting a buffer-descriptor pointer on the transmit queue in *vpmwrite*. *Ct* is the number of bytes in the buffer. When executing on a PDP11, *dp* is the pointer to the buffer descriptor; *dp* is not meaningful when executing on a VAX because pointers are four bytes on a VAX and the argument corresponding to *dp* is declared as a *short*.

vpmsave ('r', dev, cnt, dp->d_bos)

247 r 100 500 500

Called from *vpmread* just after *cnt* bytes have been moved to the user's read buffer. The parameter *dp->d_bos* is the number of bytes remaining in the current receive buffer.

vpmsave ('s', dev, vp->vb_state, 0)

248 s 100 401 0

Called just before the normal return from *vpmstart*. The parameter *vp->vb_state* contains the state bits for the interface module. For the definitions of the state bits, refer to the source file *vpmb.c*.

vpmsave ('t', dev, vp->vb_state, vp->vb_xbkmc)

249 t 100 0 0

Called just before the normal return from *vpmstop*. The parameter *vp->vb_xbkmc* is the number of transmit buffers currently held by the VPM interpreter. It can be non-zero if the protocol script or interpreter terminates in error.

vpmsave ('X', dev, vp->vb_xbkmc, 0)

250 X 100 1 0

Called from *vpmbrint*, the interface module's receive-interrupt routine, each time the VPM interpreter returns a transmit buffer.

vpmsave ('R', dev, vp->vb_vrkmc, 0)

251 R 100 1 0

Called from *vpmbrint* each time the VPM interpreter returns a receive buffer. The parameter *vp->vb_rbkmc* contains the number of receive buffers currently held by the interpreter.

vpmsave ('T', dev, sel4, sel6)

252 T 100 370 21 34

Called from *vpmbrint* when a *trace* report is received from the interpreter. This occurs when the protocol script executes a *trace* primitive. *Sel4* contains the value of the script location counter (plus two) at the time the *trace* primitive was executed. By referring to the assembly-language listing of the protocol script generated by the *-l* option of *vpmc*, the point in the protocol script at which the *trace* was executed can be determined. The value of the location counter is two greater than the location of the *trace* instruction as shown in the assembly-language listing. *Sel6* contains the byte or bytes passed by the *trace* primitive. *Vpmtrace* prints these two bytes in separate fields.

vpmsave ('E', dev, sel4, sel6)

253 E 244 21

Called from *vpmbrint* when an error-termination report is received from the interpreter. *Sel4* contains the script location counter at the time execution of the script was terminated. *Sel6* contains the termination code. For an explanation of these codes see the manual entry for *vpm*(4).

vpmsave ('P', dev, sel4, sel6)

254 P 100 2105 1055

Called from *vpmbrint* when a script report is received from the interpreter. This occurs when the protocol script executes a *rtnrpt* primitive. *Sel4* and *sel6* contain the four bytes transferred by this primitive.

vpmsave ('F', dev, sel4, sel6)

255 F 100 3 0

Called from *vpmbrint* when an error-count report is received from the interpreter. *Sel4* and *sel6* do not contain any meaningful data for this event type.

vpmsave ('S', dev, sel4, sel6)

256 S 100 401 0

Called from *vpmbrint* when a start-up report is received from the interpreter. The low-order eight bits of *sel4* contain a parameter defining the maximum number of transmit buffers the interpreter can accept; the high-order eight bits contain a parameter defining the maximum number of receive buffers. *Sel6* contains the options supported by the interpreter.

vpmsave ('C', dev, vp->vb_state, bp->vb_xbkmc)

257 C 100 1 0

Called from *vpmclean* just before the data structure associated with *dev* is initialized.

vpmsave ('O', dev, vp->vb_state, 0)

258 0 100 1 0

Called from *vpmok* if the interpreter should fail to indicate its sanity by issuing an "I'm-OK" report within the prescribed time limit.

Appendix 5: Adding VPM to a UNIX Release 3.0 System

The UNIX Release 3.0 distribution tapes contain VPM Release 2.0. This includes the compiler, drivers, interpreters, utility commands, protocol scripts, and test programs.

The *makefile vpm.mk* found in */usr/src/cmd/vpm* may be used to make and install all VPM commands.

To add the VPM and trace drivers to a UNIX 3.0 system, do the following:

1. Make sure that the following two lines appear in the file */etc/master*:

```
vpm     0   37   206   vpm   0   0     15   16   5
trace   0   35   206   tr    0   0     16   4    1
```

2. Add the following line to the file */usr/src/uts/*/cf/cfigpa* (or its equivalent):

```
vpm     0   0    0     n
```

   where *n* is the number of minor devices required. The * represents either *pdp11* or *vax*.

3. To the same file add the following line for each trace minor device:

```
trace   0   0    0     n
```

   where *n* is the number of minor devices required. Minor device 0 is used by the *vpmtrace* command and minor device 1 is used by *vpmsnap*.

4. If KMCs are being added to the system, add the following line to the same file for each KMC:

```
kmc11   vector   address   priority
```

   where *vector* is the interrupt vector location (octal), *address* is the device address (octal), and *priority* is the bus request level (normally 5).

A special file must be created in */dev* for each KMC, VPM, and trace device. To make these special files, use *mknod*(1M) as follows:

For KMCs:

```
/etc/mknod /dev/kmc? c X ?
```

where *X* is the major device number of the KMC driver as printed by **config −t** (see the manual entry for *config*(1M)*[4]*) and *?* is the minor device number that must be in the range 0 to 3.

For VPMs:

```
/etc/mknod /dev/vpm c Y Z
```

where *vpm* is a unique device name; *Y* is the major device number of the VPM driver; and *Z* is a decimal or octal number whose binary representation is defined as follows: the low-order four bits specify one of up to 16 minor devices of the standard VPM protocol module; the next two bits specify one of up to four VPM interface-module minor devices; the next two bits specify the minor device number of the KMC to be used for this special file.

For trace devices:

```
/etc/mknod /dev/trace c Y 0
/etc/mknod /dev/snap c Y 1
```

where *Y* is the major device number of the trace driver.

Hardware Installation and Switch Settings

The KMC11-B microprocessor and DMC11-DA, -FA, or -MD line unit must be installed in adjacent slots of a PDP-11 or VAX-11/780 backplane. Care should be taken not to exceed the DC power capacity of the cabinet in which the items are installed. The microprocessor and line unit are interconnected by a one-foot ribbon cable. The DMC11-DA or -FA line unit is connected to a suitable synchronous modem by a DEC-supplied modem cable. If the HDLC interpreter is used, the modem must be optioned for full-duplex (four-wire) operation; at speeds above 1200 bits per second this will normally require a private line. The DMC11-DA has an RS-232 interface that is suitable for connection to data sets such as the 208 and 209. The DMC11-FA has a CCITT V35 interface. The DMC11-MD has an integral 56 KB modem; this unit must be connected by a pair of coaxial cables to another DMC11-MD. The device address and interrupt vector address switches on the KMC should be set for the selected addresses. The KMC should also be wired for the selected bus priority (normally 5). All switches and jumpers on the DMC line unit should be in the normal configuration prescribed by the relevant DEC maintenance manual, but with one exception: the NO CRC switch (switch S2 in switch pack number 1) should be in the ON position. The purpose of this switch setting is to inhibit hardware CRC generation. Hardware CRC generation is not used with the VPM software for this device.

If the KMC is a Revision E, a DEC field change (ECO number NU007) is required before it can be used with the VPM or DZ/KMC software. If the change has already been installed, the capacitor that controls the KMC internal clock (capacitor C40, located four IC's over from the right edge of the KMC hex board—component side facing you, fingers down) will have a value of 4700 pF.

Appendix 6: Testing VPM

During the course of developing and testing VPM, a number of programs and test procedures have evolved which may prove useful to those adding VPM to a system or using VPM for the first time. These programs and procedures will help to check the correct installation and operation of the hardware and software as well as help a new user of VPM to gain familiarity with the package. These programs may be found in */usr/src/cmd/vpm/demo* and */usr/src/cmd/vpm/scripts*.

Decbin

Decbin is a simple KMC program that exercises enough of the KMC memory and instruction set so that a correct result provides reasonable assurance that the KMC is functioning properly. It does *not* exercise the interface between the KMC and the DMC11 line unit.

To run this test, you must compile file *decbin.k* in directory */usr/src/cmd/vpm/demo*. This can be done as follows:

```
/lib/cpp /usr/src/cmd/vpm/demo/decbin.k | kasb
```

You must then load and run the resulting *a.out* and then dump the KMC and its registers. The following sequence of commands will accomplish this:

```
kasb -d /dev/kmc?
.reset
.load
.run
.reset
.dump
.regs
```

The *.regs* command to *kasb* will produce a register dump similar to the following:

```
csr:   377   0   0   0   0   0   0  20
lur:     0  20   0 101   0 377 377  53
reg:     0 326  42  64   0 276   0  46
reg:   142  73 321  71 156  61 116 356
 io:   377 377 377 377 377 371 377 377
npr:     0  20   0 brg: 356   0 mem:  61
```

If the value of r5 (the sixth number in line three of the register dump) is not 276, something is wrong with the KMC hardware or the software used to load and execute programs in it.

Tset

Tset is a C program that opens a particular *vpm* device (*/dev/vpm0*) and writes a string of characters to it. It then reads the same device and compares the string of characters received to the string sent. If the two strings match, the program prints the string followed by the message "It worked!!!!!." This program will work only when a loop-back script such as *loop.r* has been loaded into the KMC. To run this test:

1. Compile *tset.c*:

   ```
   cc -o tset tset.c
   ```

2. Compile *loop.r*:

   ```
   vpmc -o loop.o loop.r
   ```

3. Load *loop.o* into the KMC:

   ```
   /etc/vpmstart /dev/kmc? 6 loop.o
   ```

4. If testing the VPM event-tracing capability, execute *vpmtrace*:

   /etc/vpmtrace > t&

5. Execute *tset*:

   tset

6. Print *t*:

   cat t

Sr

Sr opens */dev/vpm0* and forks to create a *send* process and a *receive* process. The *send* process reads up to 512 bytes at a time from its standard input and writes them to */dev/vpm0*. The receive process reads */dev/vpm0* and writes the received data to its standard output. This program may be used with the protocol script *loop.r*. The procedure for running *sr* is similar to that used with *tset*. Steps 2, 3, and 4 need not be repeated if the interpreter and *vpmtrace* are still running.

To execute *sr*:

sr < infile > outfile

The *send* process exits after it has read and transmitted the last data block of the file. The *receive* process goes into a loop that sets an alarm and reads */dev/vpm0*. If the alarm goes off before the *read* completes, the process exits.

Tcmd

Tcmd.c when used with the protocol script *tcmd.r* tests several new features of Release 2.0 of VPM: communications between a user program or a protocol module and the protocol script, reading and resetting the interpreter's error counters, and the time-stamped tracing capability. To execute *tcmd*, follow the procedures given for the first test using *tcmd.c* and *tcmd.r* in place of *tset.c* and *loop.r*. Execute *vpmsnap* instead of or in addition to *vpmtrace*.

Lapb.r

Lapb.r is the protocol script for BX.25 Level 2. To install this script in a particular KMC, proceed as follows:

```
cp /usr/src/cmd/vpm/scripts/lapb.r .
cp /usr/src/cmd/vpm/scripts/const .
cp /usr/src/cmd/vpm/scripts/tconst .
vpmc -mi hdlc -o lapb.o lapb.r
/etc/vpmstart /dev/kmc? 6 lapb.o
```

Testing this script requires two KMCs, which may be on different host computers. The KMCs must be connected by a pair of full-duplex synchronous modems or by a full-duplex synchronous null modem.[2] *Sr* should be executed simultaneously on both machines to read and write the VPM device associated with each KMC. If both KMCs are on the same host machine, it will be necessary to edit and compile a copy of *sr.c* so that it opens */dev/vpm1* instead of */dev/vpm0*. The original and modified versions of *sr* can then be executed simultaneously to exercise the two KMCs.

---

2. A suitable null modem is the Avanti 300, which is manufactured by Avanti Communications Corporation, Newport, RI.

To obtain maximum efficiency from this script, it may be necessary to modify the values of some of the parameters in the *const* file. The appropriate values for these parameters depend on the link speed and maximum frame size. Guidelines for adjusting these parameters are given in [3].

Lapbt.r

This script is identical to *lapb.r* except for some additional *trace* statements. It may be tested in the same manner as *lapb.r*. *Vpmtrace* may be used to display the trace information.

Itr.r

Itr.r is a simplified version of *lapb.r*. Unlike *lapb.r* and *lapbt.r*, this script can be exercised in a loop-back mode. To run a loop-back test, attach a DEC H-325 test connector to the end of the modem cable for the DMC11-DA line unit that is connected to the KMC11-B to be used for the test. Then compile *itr.r* and load the resulting *a.out* into the KMC using the procedure described above for *lapb.r*, substituting *itr* for *lapb*. A loop-back test can then be run using *tset* or *sr*.

January 1981

A Dial-up Network of UNIX Systems

D. A. Nowitz

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

A network of over one hundred 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. There are several features that helped make it a successful system:

- The start-up 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, but hard-wired communication lines can be used to increase speed.
- The command for sending/receiving files is simple to use.
- A general remote execution facility is part of the system; remote mail is one use of this feature.
- Adequate security facilities are available, so the site administrators feel comfortable about being on the network.

Introduction

The UNIX operating system[1] is a time-sharing system that runs on small to large minicomputers. A typical user gets access to the system through the telephone network. Since each computer is connected to the telephone network, they are potentially connected together; all that is needed to connect the machines is an automatic dialer and a program that can emulate a terminal (see Figure 1).

Many UNIX systems are used within the Bell System. Although there are some differences between them, a large set of common software modules exist: compilers, text editors, assemblers, linking loaders, debuggers, phototypesetting programs, and so on. An informal network emerged out of the need to exchange, deliver and maintain software. The telephone network provides the transmission path and the computer programs described in this paper implement the necessary protocols.

Over the past year, the network has grown to over one hundred machines throughout the country. There are several major uses of the network:

† UNIX is a trademark of Bell Laboratories.

- distribution of software
- distribution of documentation
- personal communication (mail)
- data transfer between closely sited machines
- transmission of debugging dumps and data exposing bugs
- production of hard-copy output on remote printers.

Design

In keeping with the style of UNIX commands, the user interface is quite simple. To copy a file from the local system to a remote system, the user would execute the command:

```
uucp file1 sysx!file2
```

where *file1* and *file2* are file names, and *sysx* is the remote system name. All details of the connection (e.g., device(s) used for the connection, phone number, times the remote system is available, its login sequence and password, retries for unavailable device or busy phone) are hidden from the user. The file transfer takes place when the connection is made.

We had to adapt to a community of systems that 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, obeying all local protection rules.

We distinguish between "active" and "passive" systems on the network. Active systems have an automatic calling unit or a hard-wired line to another system: they 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.

Several groups, both inside and outside Bell Laboratories, have constructed networks using hard-wired connections exclusively.[2], [3] Our network, however, uses both dial-up and hard-wired connections so that service can be provided to as many sites as possible and so the slower dial-up paths can be automatically used if a hard-wired link is out of service. Dial-up connections are made at either 300 or 1200 baud; hard-wired connections are asynchronous up to 9600 baud and might run even faster on special-purpose communications hardware.[4], [5] Thus, systems typically join our network first as passive systems. When they find the service important, 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 time, however, must users change their programs or procedures.

The basic operation of the network is simple. Each participating system has a spool directory in which work (files to be moved or commands to be executed remotely) is placed. The **UUCICO** program performs all transfers. This program is started periodically to perform the file transfers; it selects devices, establishes the connection to the remote machine, performs the required login sequence, performs file security checks, transfers data files, logs results, and notifies specified users of transfer completions.

After the calling program completes all the work for the called system, the called machine sends any files that have been queued for the calling system. In this way, all services are available from all sites; passive sites, however, must wait until called. A variety of line protocols may be used; this gives users some flexibility and provides a mechanism for distributing other protocols in the future.* As long as the caller and called programs have a protocol in common,

* There is only one known system that has implemented an additional protocol. The protocol is used for high-speed transfers using a shared disk.

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 remains in the spool directory until the destination machine can be reached.

In addition to the **UUCP** command, the user has the **UUX** command which allows execution of programs that require resources of remote machines. A common use of this facility is to format a printout on the local machine and send the result to a remote machine which has a hard copy output device. Remote mail is implemented using the **UUX** command but its execution is embedded in the standard *mail* command.

Processing

The user has two commands that 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 the **UUCICO** program. Figure 2 shows the major blocks of the file transfer process.

The file names in the spool directory are constructed to allow the **UUCICO** program to identify the work and data files, the remote machines that should be called, and the order that the files for a particular system should be processed.

The call is made using information from several files. A single conversation between a pair of systems is ensured by the use of a lock file. A "systems" file contains information for making a connection to a remote machine:

[1] system name,

[2] system access time (days-of-week and times-of-day),

[3] device or device type to be used for the call,

[4] line speed,

[5] phone number,

[6] login information (multiple fields).

The *phone number* may contain abbreviations (e.g. "nyc", "boston") that 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. If the connection fails after two attempts, an alternate path can be tried. (The presence of more than one entry for a system in the "systems" file indicates alternate paths.) If the connection is complete, the *login information* is used to log into the remote system. The conversation between the two **UUCICO** programs begins with a handshake started by the called (or *SLAVE*) system. The *SLAVE* sends a message to let the *MASTER* know that 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 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 no common protocol.

During the processing, one program is the *MASTER* and the other is the *SLAVE*. Initially, the calling program is the *MASTER*. These roles may switch one or more times during the conversation.

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,
X get files whose names are determined on the remote system,
C copy complete,
H hangup.

The *MASTER* uses the first three to request file transfers. The *SLAVE* responds to each with a *yes* or *no*.

The send, receive and execute replies are based on permission to access the requested file/directory. After each file is copied to the receiving system, a copy-complete message is sent by the receiver of the file. The requests and results are logged on both systems, and, if requested, mail is sent to the user reporting completion. A failure message is sent by mail to the requester.

The *MASTER* executes all the work for the remote, followed by an *H* message. The *SLAVE* checks its spool directory for work. If work for the remote system exists, an *HN* message is sent and the programs switch roles; otherwise, an *HY* is sent. When the *MASTER* receives an *HY* message, it echoes it back to the *SLAVE* and the protocols are turned off. Each program sends a final *OO* (close) message to the other. Figure 3 shows a sample conversation.

Security

The implementation of this network between independent sites, many of which store proprietary programs and data, illustrates the pervasive need for security and administrative controls over file access. A number of security features evolved during the development of the system:

- file directory access restrictions,
- file monitoring,
- call back,
- call sequence checking,
- limited commands for the **UUCP** logins,
- restricted commands available to the **UUX** command,
- limited access to remote login information.

Each site, when configuring its programs and system files, limits and monitors transmissions. The administrator can give some remote systems limited file access while others have the same access privileges as the local users. Each system establishes a public directory for the **UUCP** program. A degree of file security can be achieved if the administrator allows the remote **UUCP** programs to access only this public directory. This requires a local user for remote sites to get or send files. Records are kept identifying all files that are moved into and out of the local system, and also of how the requester of such accesses identified themselves.

A site can arrange to permit users to call up and request that work be done; the calling users are then called back before the work is actually performed. This makes it possible to verify that the requester is legitimate. Masquerading is difficult even if the necessary password is known.

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. A would-be impersonator must steal not only the correct phone number, user name, and password, but also the sequence count, and must call in promptly before the next legitimate request from either side. Even a successful masquerade will be detected on the next correct conversation.

The "systems" file, which is described in the Processing Section, contains information to allow the **UUCICO** program to login to the remote machines. This information would usually permit almost complete access to the system. The normal file system protections are used to restrict access of the "systems" file to the **UUCP** programs and the administrator. This gives some security, but it depends on the remote system administrator. To minimize this problem, we set up the system so that the only program that can be executed with the **UUCP** login is the **UUCICO** program. The system administrator can use the directory access restrictions to protect the local system without depending on a remote system to protect the login information.

The **UUX** command allows users to execute commands on remote systems. To protect a system, the administrator has to specify a list of commands that the **UUCP** system can execute. All commands received are checked against the list.

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 pwba!dan" the mail is sent to user "dan" on system "pwba".

The primary uses of our network to date have been in software maintenance. New programs (or new program versions) are sent to users, and potential bugs are returned to authors. A "stockroom" has been established at two sites. This allows remote users to call in and request software without bothering the author. A "stock list" of available programs, new bug fixes, and utilities is updated regularly.

Test cases are retrieved 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.*

The **UUX** command has been useful for providing remote output. There are some machines that do not have hard-copy devices, but that are connected over 9600 baud communication lines to machines with printers. The **UUX** command allows the formatting of a printout on a local machine and printing on a remote machine using standard UNIX commands.

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 at 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 fast links.

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 a few 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. During a typical week for a group a three co-located systems, 30 users made about 300 requests resulting in the transfer of about 3 million bytes. (These transfers took place over 9600 baud hard-wired lines.) On a system that distributes and

* 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.

maintains standard system software, a typical week consists of transferring about 1500 files (10 million bytes): this includes the dial-up network at 300 or 1200 baud and hard-wired local lines.

Presently, the total number of sites in the network is about 120. This includes most of the Bell Laboratories full-size machines that run the UNIX operating system, many operating telephone companies, some Western Electric sites, and several universities. Geographically, the machines range from Andover, Massachusetts to Berkeley, California.

Further Goals

Eventually, we would like to develop a full system of remote software maintenance. Conventional maintenance (a support group that mails tapes) has many well-known disadvantages.[6] 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 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" provides 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 permitting 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 by lower counseling and distribution costs.

Some slow-speed network services have been implemented. We now have inter-system *mail* 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 multiplexed channels so that long file transfers do not lock out short inquiries.

Lessons

What follows is a summary of the lessons we learned in building this system.

1. By starting the network in a way that requires no hardware or operating system changes, one 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 similar to standard UNIX system commands so that little training cost is involved.
4. In the first version, we made the mistake of using dial-up communications exclusively. The second implementation of the system permits the use of different connecting fabric, such as hard-wired, asynchronous lines: this adds flexibility to the network.

5. Security presented a bigger problem than we anticipated. We had to give the administrators features that enabled them to protect their systems. These features, however, made it difficult to do useful work. The creation of a public directory on each system alleviated some of the problem, but the casual users of **UUCP** are often unpleasantly surprised when their requests are rejected by remote systems.

Acknowledgements

We thank G. L. Chesson for his design and implementation of the packet driver line protocol, and A. S. Cohen, A. G. Fraser, 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.* **57**(6), pp. 1905-1929 (1978).

[2] T. A. Dolotta, R. C. Haight, and J. R. Mashey, "UNIX Time-Sharing System: The Programmer's Workbench," *Bell Sys. Tech. J.* **57**(6), pp. 2177-2200 (1978).

[3] G. L. Chesson, "The Network UNIX System," *Operating Systems Review* **9**(5), pp. 60-66 (1975). Also in *Proc. 5th Symp. on Operating Systems Principles,* 1975.

[4] A. G. Fraser, "Spider—An Experimental Data Communications System," *Proc. IEEE Conf. on Communications,* p. 21F (June 1974). IEEE Cat. No. 74CH0859-9-CSCB.

[5] A. G. Fraser, "A Virtual Channel Network," *Datamation,* pp. 51-56 (February 1975).

[6] F. P. Brooks, Jr., *The Mythical Man-Month,* Addison-Wesley, Reading, MA (1975).

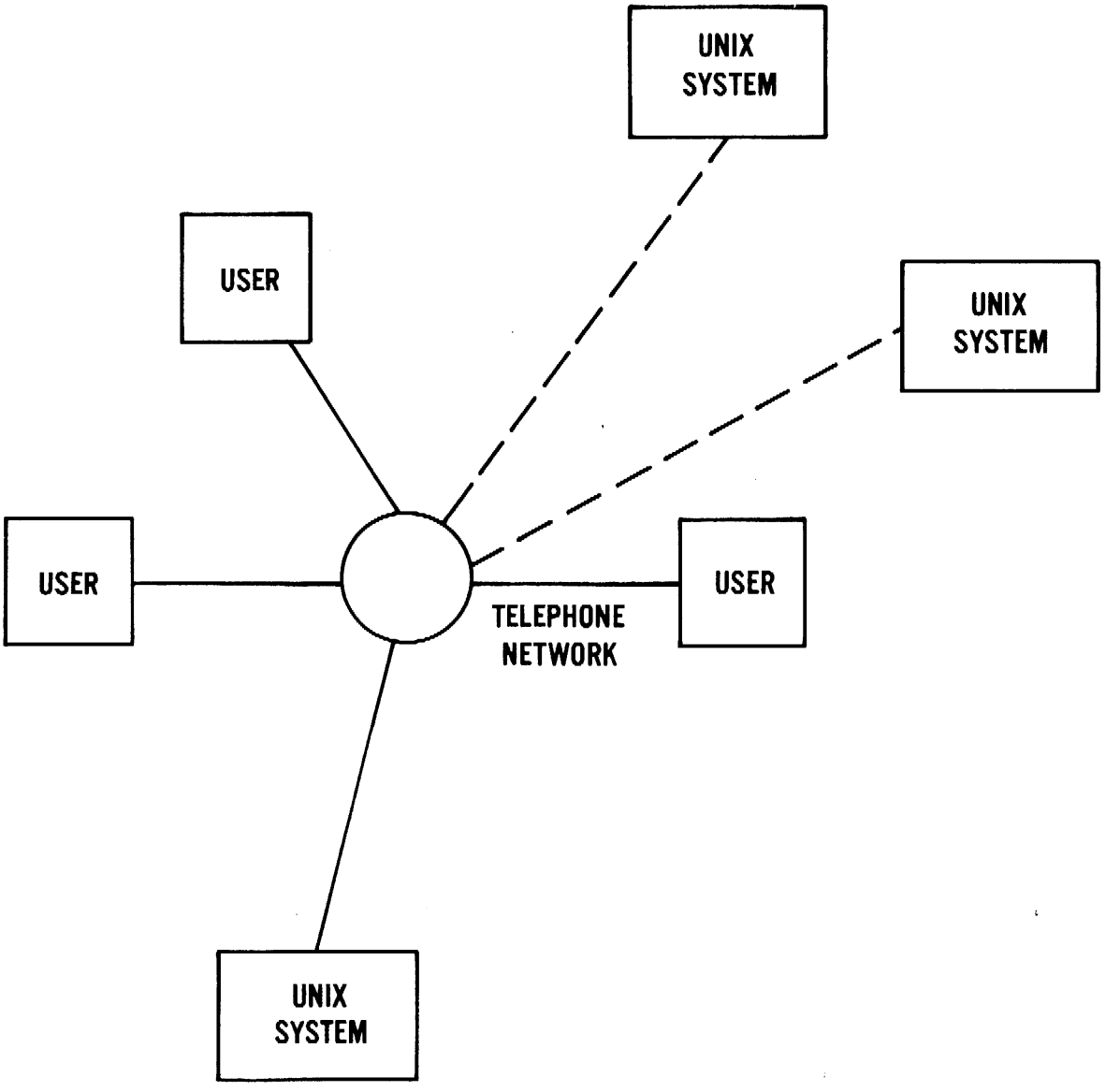


FIGURE 1

UUCP NETWORK - ACCESS TO REMOTE UNIX SYSTEM THROUGH DIALUP TELEPHONE NETWORK

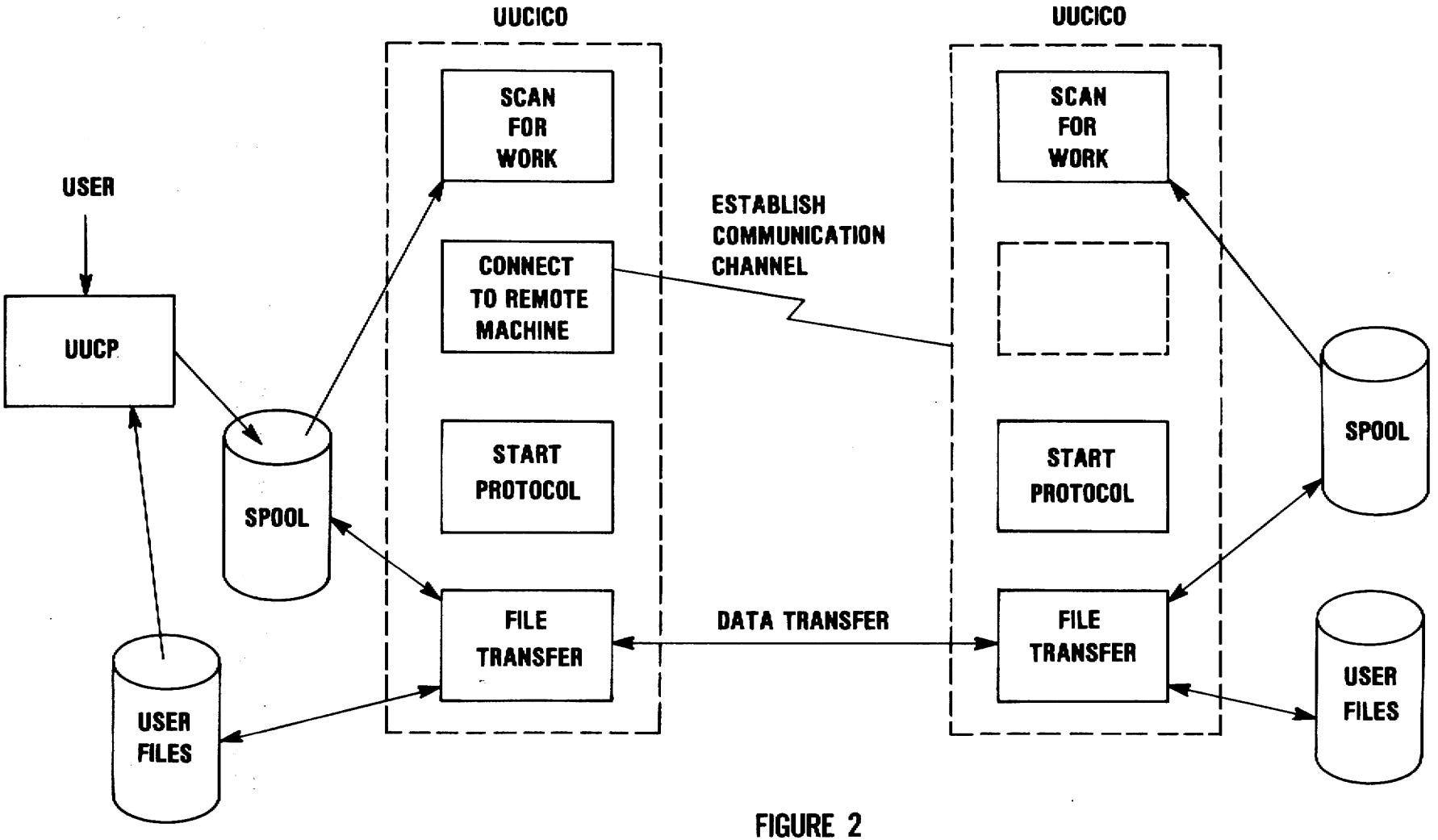


FIGURE 2
FILE TRANSFER PROCESS

CALLING SYSTEM	MESSAGES & DATA	CALLED SYSTEM
	555-6789 ⟶	[Dial up]
⟵	**login:**	
	user-id, password ⟶	[Identification]
⟵	**S**	[Handshake and
	S 36 myname ⟶	sequence check]
⟵	**P** a, b, c	[Offer protocols]
	U b ⟶	[Choose protocol]
	S /abc/def/ghi ⟶	[Send file]
⟵	**SY**	[OK to send]
═	DATA ⟹	
	R /abc/def/jkl	[Get file]
⟵	**RY**	[OK to receive]
⟸	DATA ═	
	H ⟶	[Offer to stop]
⟵	**HN**	[Refused]
⟵	**S** /abc/mno/prs	[Called system sends]
	SY ⟶	[OK to send]
⟸	DATA ═	
⟵	**H**	[Offer to stop]
	HY ⟶	[Accepted]
⟵	**(OO)**	[Close]
	(OO) ⟶	
	Hangup	

FIGURE 3
SAMPLE CONVERSATION

UUCP Implementation Description

D. A. Nowitz

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Uucp is a series of programs designed to permit communication between UNIX† systems using either dial-up or hard-wired communication lines. This document gives a detailed implementation description of the current implementation of uucp. Is is designed for use by an administrator/installer of the system. It is not meant as a user's guide.

Introduction

Uucp is a series of programs designed to permit communication between UNIX systems using either dial-up or hard-wired communication lines. It can be 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 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. *Execute files* are scripts for UNIX commands that involve the resources of one or more systems.

There are four primary programs:

uucp	builds *work files* and gathers *data files* in the spool directory for data transmission.
uux	creates *work files*, *execute files*, and gathers *data files* for the remote execution of UNIX commands.
uucico	executes the work files for data transmission.
uuxqt	executes the scripts for UNIX command execution.

There are a couple of administrative programs:

uulog	gathers temporary log files that may occur due to lockout of the uucp log file and reports some information such as copy requests and completion status.
uuclean	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.

---

† UNIX is a trademark of Bell Laboratories.

1. M. E. Lesk and A. S. Cohen, UNIX *Software Distribution by Communication Link*, private communication.

1. Uucp—UNIX to UNIX File Copy

The *uucp* command is the user's primary interface with the system. The command is 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 where the file or files reside or where they will be copied.

Uucp has several options:

- −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.
- −e*sys* Send this job to system *sys* to execute. (Note that this will only work when the system *sys* allows *uuxqt* to execute a *uucp* command. See the "Uuxqt" and "Security" sections.)
- −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 to the requester on completion of the work.
- −n*user* Notify *user* on the remote machine that a file has been sent.

There are several options available for debugging:

- −r Queue the job but do not start *uucico* program.
- −x*num* *Num* is a level number between 1 and 9; higher numbers give more debugging output.

The destination may be a directory name, in which case the file name is taken from the last part of the source's name. If the directory exists, it must be writable by everybody. (Note that if the destination is a directory name and the "−d" option is specified to create the directory, the directory name must be followed by "/".) The source name may contain special shell characters such as "?*[]". These will be expanded on the appropriate 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 of *user* on the specified system. File names beginning with "~/" translate into the public directory (usually /usr/spool/uucppublic) on the remote system. For names with partial path-names, the current directory is prepended to the file name. File names with ../ are not permitted for security reasons.

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, the system-part of each argument, and the options to classify the work into several types:

[1] Copy source to destination on local system.

[2] Receive files from other systems.

[3] Send files to a remote system.

[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.

[6] Request that the *uucp* command be executed by a remote system.

After the work has been set up in the spool directory, the *uucico* program is started to try to contact the other machine and execute the work (unless the −r option was specified).

Type 1 - local copy

The copy is done locally. The −*m* and −*n* options are not honored in this case.

Type 2 - receive files

A *work file* is created or appended with a one line entry for each request. The upper limit to the number of files per *work file* is set in *uucp.h*. (The default setting is 20.) After the limit has been reached, a new *work file* is created. (All *work files* and *execute files* use a blank as the field separator.) The fields for these entries are given below.

[1] R

[2] The full path-name of the source or a ~something/path-name. The ~*something* part will be expanded on the remote system.

[3] The full path-name of the destination file. If the ~*something* notation is used, it will be immediately expanded.

[4] The user's login name.

[5] A "−" followed by an option list. The options −m and −d may appear.

Type 3 - send files

Each 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 for these entries are given below.

[1] S

[2] The full-path name of the source file.

[3] The full-path name of the destination or ~something/file-name.

[4] The user's login name.

[5] A "−" followed by an option list. The options −d, −m, and −n may appear.

[6] The name of the *data file* in the spool directory. A dummy name, "D.0" is used when the −c option is specified.

[7] The file mode bits of the source file in octal print format (e.g., 0666).

[8] The user on the remote system to be notified upon completion of the file copy when the "−n" option is specified.

Type 4 and Type 5 - remote uucp required

Uucp generates a *uucp* command and sends it to the remote machine; the remote *uucico* executes the *uucp* command.

Type 6 - remote execution

This occurs when the "−e" option is used. In this case, the *uux* facility is used to create and send the request. This requires that the remote *uuxqt* program allows 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 that do not contain a "!" will not be treated as files. (They will not be copied to the execution machine.) An argument that contains a "!" but is not to be treated as a file at the present time, can be escaped by using "()" around the argument. (Note that the "()" symbols must usually be escaped with a "\" symbol.) The "−" is used to indicate that the standard input for *command-string* should be inherited from the standard input of the *uux* command. The following options are available for debugging:

−r Don't start *uucico* or *uuxqt* after queuing the job.

−x*num* *Num* is a level number between 1 and 9; higher numbers give more debugging output.

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* that 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 send command (type 3 above).

For required files that 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 for execution by the *uucico* program.

The *execute file* contains a script that will be processed by the *uuxqt* program. It is made up of several lines, each of which contains an identification character and one or more arguments. The lines are 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 a unique name used for file transmission 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. The *uuxqt* program will check for the existence of all these 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. (Note that if there is a standard input specified, it will also appear in an "F" line.)

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 that 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 (usually /usr/lib/uucp/.XQTDIR) 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. (Note that a check is made to see that the command is allowed as specified in the *uuxqt* program.) After execution, the standard output is copied or sent to the proper place.

3. Uucico—Copy In, Copy Out

The *uucico* program will perform several 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 demon specified in a crontab entry,

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 logins used by remote systems to access uucp.)

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.

Uucico 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 that 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 a level number between 1 and 9; higher numbers give more debugging output.

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, zero 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 that reside in the uucp program directory (usually /usr/lib/uucp). At the start of the call process, a lock is set to forbid multiple conversations between the same two systems.

The *L.sys* file contains information required to make the remote connection:

[1] system name,

[2] times to call the system (days-of-week and times-of-day) and the minimum time delay before retry,

[3] device or device type to be used for call,

[4] line class (this is the line speed on almost all systems),

[5] phone number if field [3] is *ACU* or the device if not *ACU*,

[6] login information (zero or more 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) that 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 each devices that satisfy [3] and [4] until a call is made, or no more devices can be tried. If a device is successfully opened, a lock file is created. If the call is completed, 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 *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 *MASTER* program does a work search similar to the one used in the "Scan For Work" section. (The *MASTER* has been specified by the "−rl" uucico option.) Each message used during the work processing is specified by the first character of the message:

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 for the remote system is complete, at which point an *H* message will be sent. The *SLAVE* will reply with *SY*, *SN*, *RY*, *RN*, *HY*, *HN*, *XY*, or *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 spool directory to the destination. Otherwise, a *CN* message is sent. (In this case, the file is put in the public directory, usually /usr/spool/uucppublic, and the requester is notified by mail.) The requests and results are logged on both systems.

The hangup response is determined by a work scan of the *SLAVE*'s spool directory. If work for the remote system exists 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 unless a "−s" option was specified.

4. Uuxqt−Uucp Command Execution

The *uuxqt* program is used to execute scripts generated by *uux*. The *uuxqt* program may be started by either the *uucico* or *uux* programs or a demon specified by a *crontab* entry. 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 is verified and executed.

The *execute file* is described in the "Uux" section above.

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

When a *uucp* program can not make a log entry directly into the *LOGFILE* an individual log file is created: a file with prefix *LOG*. This will sometimes occur when more than one uucp process is running. Periodically, *uulog* may be executed to append these files to the *LOGFILE*.

The *uulog* program may also be used to request the output of *LOGFILE* entries. The request is specified by the use of the 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 uucp daily demon. Its function is to remove files from the spool directory that are more than 3 days old. These are usually files for work that can not be completed. The requester of this work is notified that the files have been deleted.

There are several options:

- −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. This mail is sometimes useful for administration.)
- −n*hours* Change the aging time from 72 hours to *hours* hours.
- −p*pre* Examine files with prefix *pre* for deletion. (Up to 10 of these options 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 out/in any file that is readable/writable by a 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 administrator of the *uucp* system.

- The login for uucp does not get a standard shell. Instead, the *uucico* program is started so that all work is done through *uucico*.
- The owner of the uucp programs should be an administrative login. It should not be one of the logins used for remote system access to uucp.
- 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 of the caller's identity.
- The *uuxqt* program comes with a list of commands that it will execute. A "PATH" shell statement is prepended to the command line as specified in the *uuxqt* program. The installer may modify the list or remove the restrictions as desired.
- The *L.sys* file should be owned by the uucp administrative login and have mode 0400 to protect the phone numbers and login information for remote sites.

– The programs *uucp*, *uucico*, *uux*, *uuxqt*, *uulog*, and *uuclean* should be owned by the uucp administrative login, have the setuid bit set, and have only execute permissions.

8. Uucp Installation

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 or the uucp administrative login. However, several remote computers may use the same login name.

Each computer should be given a unique *system name* that is transmitted at the start of each call. This name identifies the calling machine to the called machine. The *login/system* names are used for security as described later in the *USERFILE* section.

There are several source modifications that may be required before the system programs are compiled. These relate to the directories, local system name, and attributes of the local environment.

There are several directories used by the uucp system:

lib	(/usr/src/cmd/uucp) – This directory contains the uucp system source files.
program	(/usr/lib/uucp) – This is the directory used for some of the executable system programs and the system files. Some of the programs reside in "/usr/bin".
spool	(/usr/spool/uucp) – This is the uucp system spool directory.
xqtdir	(/usr/lib/uucp/.XQTDIR) – This directory is used during execution of the *uux* scripts.

The names in parentheses above are the default values for the directories. The italicized names *lib, program, xqtdir,* and *spool* will be used in the following text to represent the appropriate directory names.

There are two files that may require modification, the *makefile* file and the *uucp.h* file. (On some systems, the makefile is named *uucp.mk*.) In addition, the "uuxqt.c" program may be modified as indicated in the "Security" section above. The following paragraphs describe the modifications.

uucp.h modification

Several manifests in "uucp.h" may need modification for the local system environment:

UNAME	should be defined if the "uname" function is available.
MYNAME	should be modified to the name of the local system if UNAME is *not* defined.
ACULAST	is the character required by the ACU as the last character. For most systems, it is a "–".
DATAKIT	should be defined if the system is on a datakit network.
DIALOUT	should be defined if the "C" library routine "dialout" is available.

makefile modification

There are several *make* variable definitions that may need modification:

INSDIR	is the *program* directory (e.g., INSDIR=/usr/lib/uucp). This parameter is used if "make cp" or "make install" is used.
IOCTL	is required to be set if the "ioctl" routine is *not* available in the standard "C" library; the statement "IOCTL=ioctl.o" is required in this case.
PUBDIR	is a public directory for remote access. This is also the login directory for remote uucp users. It should be the same as that defined in "uucp.h".

SPOOL is the uucp spool directory. This should be the same as that defined in "uucp.h".

XQTDIR is the directory for uuxqt to use for command execution. It is also defined in "uucp.h".

OWNER is the administrative login for uucp.

Compile the system

The command

make install

will make the required directories, compile all programs, set the proper file modes, and copy the programs to the proper directories. This command should be run as *root*. The command

make

will compile the entire system.

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 that are required for execution. They should reside in the *program* directory. The field separator for all files is a space.

L-devices

This file contains call-unit device and hard-wired connection information. The special device files are assumed to be in the */dev* directory. The format for each entry is

type line call-unit speed

where

type is a device type such as ACU or DIR. The field can also be used to specify particular ACUs for some calls by using a suffix on the ACU field, e.g., ACU3. This names should be used in *L.sys*.

line is the device for the line (e.g., cul0).

call-unit is the automatic call unit associated with *line* (e.g., cua0). Hard-wired lines have a number "0" in this field.

speed is the line speed.

The line

ACU cul0 cua0 300

would be set up for a system that has device "/dev/cul0" wired to a call-unit "/dev/cua0" for use at 300 baud.

L-dialcodes

This file contains the dial-code 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.

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 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 *sys*.

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 on the first line in the *USERFILE* that has 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 on the first line in the file that has the system name that matches 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". (Note that this type restriction is seldom used.)

The lines

```
u,m /usr/xyz /usr/spool
u, /usr/spool
```

allows any remote machine to login with name *u*. If its system name is not *m*, it can only ask to transfer files whose names start with "/usr/spool". If it is system *m*, it can send files from paths "/usr/xyz" as well as "/usr/spool".

The lines

```
root, /
, /usr
```

allow any user to transfer files beginning with "/usr" but the user with login *root* can transfer any file. (Note that any file that is to be transferred must be readable by anybody.)

L.sys

Each entry in this file represents one system that can be called by the local uucp programs. More than one line may be present for a particular system. In this case, the additional lines represent alternative communication paths that will be tried in sequential order. The fields are described below.

system name

The name of the remote system.

time

This is a string that 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 allowed for the call. Note that a time range that spans 0000 is permitted, for example, 0800-0600 means all times are allowed other than times between 6 and 8 am.

An optional subfield is available to indicate the minimum time (minutes) before a retry following a failed attempt. The subfield separator is a ",". (e.g., Any,9 means call any time but wait at least 9 minutes after a failure has occurred.)

device

This is either *ACU* or the hard-wired device to be used for the call. For the hard-wired case, the last part of the special file name is used (e.g., tty0).

class

This is usually the line speed for the call (e.g., 300). The exception is when the "C" library routine "dialout" is available in which case this is the dialout class.

phone

The phone number is made up of an optional alphabetic abbreviation and a numeric part. The abbreviation should be one that appears in the *L-dialcodes* file (e.g., mh5900, boston995−9980). For the hard-wired 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] ...

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. (e.g., login--login will expect *login*; if it gets it, the program will go on to the next field; if it does not get *login*, it will send *null* followed by a new line, then expect *login* again.)

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 number from 1 to 9 may follow the *BREAK* for example, *BREAK1* will send 1 null character instead of the default of 3. Note that *BREAK1* usually works best for 300/1200 baud lines.

A typical entry in the L.sys file would be

sys Any ACU 300 mh7654 login uucp ssword: word

The expect algorithm match all or part of the input string as illustrated in the password field above.

9. Administration

This section indicates some events and files that must be administered for the uucp system. Some administration can be accomplished by *shell files* 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 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 manually.

TM − temporary data files

These files are created in the *spool* directory while a file is 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. After the entire file is received, the *TM* file is moved/copied to the requested destination. If processing is abnormally terminated 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

program/uuclean −pTM

will remove all *TM* files older than three days.

LOG − log entry files

During execution, log information is appended to the *LOGFILE*. If this file is locked by another process, the log information is placed in individual log files which will have prefix *LOG*. These files should be combined into the *LOGFILE* by using the *uulog* program. This program will append the *LOGFILE* with the individual log files. 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.

The *LOG.* files are created initially with mode 0222. If the program that 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 into the *LOGFILE*.

STST – system status files

These files are created in the spool directory by the *uucico* program. They contain information such as login, dial-up or sequence check failures or will contain a *TALKING* status when two machines are conversing. The form of the file name is

STST.sys

where *sys* is the remote system name.

For ordinary failures, such as dial-up or login, the file will prevent repeated tries for about 55 minutes. This is the default time; it can be changed on an individual system basis by a subfield of the time field in the *L.sys* file. For sequence check failures, the file must be removed before any future attempts to converse with that remote system.

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 device. 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 (usually only on system crashes). They will be ignored (re-used) after 1.5 hours. When runs abort and calls are desired before the time limit, the lock files should be removed.

ERRLOG – uucp system error file

This file is created in the *spool* directory to record uucp system errors. Entries in this file should be rare. The messages come from the *ASSERT* statements in the various programs. Wrong modes on files or directories, missing files, and read/write system call failures on the transmission channel may cause entries in the *ERRLOG* file.

Shell Files

The *uucp* program will spool work and attempt to start the *uucico* program, but *uucico* will not always be able to execute the request immediately. 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

```
/usr/bin/uulog
program/uucico  -r1  -sinter
program/uucico  -r1
```

The "−r1" option is required to start the *uucico* program in *MASTER* mode. The "−s" option can be used for polling as illustrated in the second line where machine *inter* is being polled. The third line will process all other spooled work.

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 that 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 that 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

Two or more logins should be set up for *uucp*. One should be an administrative login: the owner of all the uucp programs, directories and files. All others are used by remote systems to access the uucp system. Each of the "/etc/passwd" entries for the *access* logins should have "*program*/uucico" as the shell to be executed. The login directory should be the public directory (usually /usr/spool/uucppublic). The various *access* login names are used in the *USERFILE* to restrict file access.

File Modes

The programs *uucp*, *uux*, *uucico*, *uulog*, *uuclean* and *uuxqt* should be owned by the *uucp* administrative login with the "setuid" bit set and only execute permissions (e.g., mode 04111). The *L.sys*, *SQFILE* and the *USERFILE*, which are put in the *program* directory should be owned by the *uucp* administrative login and set with mode 0400. The mode of *spool* should be "0755". The mode of *xqtdir* should be "0777". The *L-dialcodes* and the *L-devices* files should have mode 0444.

January 1981

The Implementation of the LP Spooling System

J. R. Kliegman

Bell Laboratories
Piscataway, New Jersey 08854

1. INTRODUCTION

LP is a system of commands that performs diverse spooling functions under the UNIX† operating system. Because its primary application is off-line printing, this paper focuses mainly on spooling to line printers. LP allows administrators to customize the system to spool to a collection of line printers of any type and to group printers into logical classes in order to maximize the throughput of the devices. Users are provided the capabilities of queuing and canceling print requests, preventing and allowing queuing to and printing on devices, starting and stopping LP from processing requests, changing the configuration of printers and finding the status of the LP system. This memo describes the implementation of LP and suggests how it can be used as a general purpose spooler.

The remainder of this paper is organized as follows: Section 2 presents an overview of the features of LP and defines terms that will be used throughout the memo. See [1] for a detailed description of the role of an LP Administrator. Section 3 tells how to build an LP system. Section 4 describes the LP directory structure and file formats. The internals of the LP scheduler are outlined in Section 5. Section 6 addresses the issue of using LP for general purpose spooling, Section 7 discusses possible extensions to LP and the last section summarizes the features that separate LP from other spooling systems.

2. OVERVIEW OF LP FEATURES

2.1 Definitions

We will define several terms before presenting a brief summary of LP commands. LP was designed with the flexibility to meet the needs of users on different UNIX systems. Changes to LP's configuration (see below) are performed by the *lpadmin*(1M) command. (A parenthesized number immediately following a command name refers to that section of the *UNIX User's Manual.*)

LP makes a distinction between printers and printing devices. A *device* is a physical peripheral device or a file and is represented by a full UNIX path name. A *printer* is a logical name that represents a device. At different points in time, a printer may be associated with different devices. A *class* is a name given to an ordered list of printers. Every class must contain at least one printer. Each printer may be a member of zero or more classes. A *destination* is a printer or a class. One destination may be designated as the *system default destination*. The *lp*(1) command will direct all output to this destination unless the user specifies otherwise. Output that is routed to a printer will be printed only by that printer, whereas output directed to a class will be printed by the first available class member.

Each invocation of *lp* creates an output *request* that consists of the files to be printed and options from the *lp* command line. An *interface program* which formats requests must be supplied for each printer. The LP scheduler, *lpsched*(1M), services requests for all destinations by routing requests to interface programs to do the printing on devices. An LP *configuration* for a system consists of devices, destinations and interface programs.

---

† UNIX is a trademark of Bell Laboratories.

2.2 Commands

2.2.1 Commands for General Use

Lp(1) is used to request the printing of files. It creates an output request and returns a *request id* of the form:

dest−seqno

to the user, where *seqno* is a unique sequence number across the entire LP system and *dest* is the destination where the request was routed.

Cancel is used to cancel output requests. The user supplies request ids as returned by *lp* or printer names, in which case the currently printing requests on those printers are canceled.

Disable prevents *lpsched* from routing output requests to printers.

Enable(1) allows *lpsched* to route output requests to printers.

2.2.2 Commands for LP Administrators

Each LP system must designate a person or persons as LP Administrator to perform the restricted functions listed below. Either the super-user or any user who is logged into UNIX as "lp" qualifies as an LP Administrator. All LP files and commands are owned by lp, except for *lpadmin* and *lpsched*, which are owned by root.

Lpadmin(1M) modifies the LP configuration. Many features of this command cannot be used when *lpsched* is running.

Lpsched(1M) routes output requests to interface programs which do the printing on devices.

Lpshut stops *lpsched* from running. All printing activity is halted, but the other LP commands may still be used.

Accept(1M) allows *lp* to accept output requests for destinations.

Reject prevents *lp* from accepting requests for destinations.

Lpmove moves output requests from one destination to another. Whole destinations may be moved at once. This command cannot be used when *lpsched* is running.

3. BUILDING LP

All LP commands are built from source code that resides in the **/usr/src/cmd/lp** directory including the make file, **lp.mk**. All structures and constants that are mentioned below are defined in the header files **lp.h** and **lpsched.h** in the same directory. Unless some of the definitions in **lp.mk** are changed, LP may be installed only by the super-user. Before installing a new LP system, make sure there is a login called **lp** on your system and that the spool directory, **/usr/spool/lp**, does not exist. **Lp**'s login directory may be **/usr/spool/lp** for convenience. To install LP, perform the following:

```
cd /usr/src/cmd/lp
make −f lp.mk install
```

This builds all LP commands and creates the directory structure which is described in the next section. The initial LP configuration produced by the preceding commands consists of no printers, classes or default destination. LP must be configured by an LP Administrator using the *lpadmin* command in order to create a useful spooler.

In addition, add the following code to /etc/rc:

```
rm -f /usr/spool/lp/SCHEDLOCK
/usr/lib/lpsched
echo "LP scheduler started"
```

This starts the LP scheduler each time that UNIX is restarted.

Several variables in **lp.mk** may be changed before installing LP to customize the system:

Variable	*Default Value*	*Meaning*
SPOOL	/usr/spool/lp	spool directory
ADMIN	lp	logname of LP Administrator
GROUP	bin	group that owns LP commands and data
ADMDIR	/usr/lib	administrator commands reside here
USRDIR	/usr/bin	user commands reside here

If an existing LP spool directory is corrupted (but not the LP programs) or if it needs to be rebuilt from scratch, make sure that *lpsched* is not running and perform the following as super-user:

1. Make copies of any interface programs that are not standard LP software. DO NOT make these copies underneath the spool directory. The path name for printer p is /usr/spool/lp/interface/p.
2. rm -fr /usr/spool/lp
3. make -f lp.mk new

This recreates the bare LP configuration described above.

WARNINGS:

1. Some LP commands invoke other LP commands. Moving them after they are built will cause some commands to fail.
2. The files under the SPOOL directory should be modified *only by LP commands*.
3. All LP commands require set-user-id permission. If this is removed, the commands will fail.

4. DIRECTORY STRUCTURE AND FILE FORMATS

The LP directory structure, as depicted in Figure 1, shows all directories and files that are under the spool directory, **/usr/spool/lp**. Section numbers in Figure 1 refer to the section numbers in this memo in which the appropriate file is described. The notation <x> means "zero or more files of type **x**".

4.1 FIFO

FIFO is a fifo (named pipe) special file where all commands send messages to *lpsched*. Any of the LP commands may write to FIFO, but only *lpsched* may read it. A subroutine named **enqueue** sends a message and its arguments to *lpsched* on FIFO. The usage of enqueue is:

```
enqueue(msg, arglist)
char msg;
char *arglist;
```

All messages are defined mnemonically in the LP header file, **lp.h**. *Arglist* is a (possibly null) blank-separated list of arguments associated with the message *msg*. Enqueue returns non-zero if *lpsched* is running and zero if not. Table 1 lists the legal messages to *lpsched*.

File Name	*Section*
spool directory	4.
<lock files>	4.8
<log files>	4.3
FIFO	4.1
class	4.9
<class files>	
default	4.2
interface	4.10
<interface programs>	
member	4.11
<member files>	
model	4.12
<model programs>	
outputq	4.4
pstatus	4.5
qstatus	4.6
request	4.13
<request directories>	
<request files>	
<data files>	
seqfile	4.7

Figure 1. LP Directory Structure

4.2 Default

The **default** file contains the name of the system default destination terminated with a new-line. If this file is absent or empty, the system has no default destination.

4.3 Log Files

The **log** file is a record of *lpsched* errors and printing activity since the time when *lpsched* was last invoked. **Oldlog** contains the same information from the previous invocation of *lpsched*.

The first (last) line of the log indicates the time that *lpsched* was started (stopped). Error messages have the form:

lpsched: error-message

For each output request that has printed (or is currently printing) there is a line with the following tab-separated fields: request id, logname of requester, printer which serviced the request and the date and time when printing began. There is more than one entry in the log for requests that were restarted after they were partially printed.

4.4 Outputq

The binary file **outputq** is a queue of output request entries that are made by the *lp* command and have the form shown in Figure 2. There is one entry for each pending or partially printed request in addition to the "deleted" entries for output requests that have been serviced since *lpsched* was last invoked. The requests for each printer are serviced strictly on a first in first out basis. **Outputq** entries are marked "deleted" by the *cancel*, *disable* and *lpsched* commands and may be modified by the *lpmove*, *disable* and *lpsched* commands.

TABLE 1. Messages Recognized by *lpsched* on FIFO

MESSAGE	*MEANING TO LPSCHED*
F_ENABLE pr	Printer **pr** has been enabled. Pending requests (if any) will be printed on **pr**.
F_NOOP	No-op to check if *lpsched* is running.
F_DEV pr path	New device for printer **pr** is **path**.
F_STATUS	This causes *lpsched* to dump internal status to the **log** file (see *Log Files*).
F_DISABLE pr	Printer **pr** has been disabled. If it is busy, printing on **pr** will stop. If another printer can service the aborted request, then it will start printing it in its entirety.
F_CANCEL dest seqno	Request id **dest−seqno** has been canceled. If it is currently printing, then printing will stop.
F_NEWLOG	This causes *lpsched* to create a new **log** file (see *Log Files*). The old log file is renamed **oldlog**.
F_REQUEST dest seqno user	Output request id **dest−seqno** has been made by **user**. If there is a printer than can service it, it will be printed immediately.
F_QUIT	This causes *lpsched* to stop running. All printing is terminated.
F_MORE pr	Printer **pr** is ready to print more requests.
F_ZAP pr	Busy printer **pr** has been disabled and its request has been canceled.

```
struct outq {
        char o_dest[DESTMAX+1];         /* output destination (class or member) */
        char o_logname[LOGMAX+1];       /* logname of requester */
        int o_seqno;                    /* sequence # of request */
        long o_size;                    /* size of request -- # of bytes of data */
        char o_dev[DESTMAX+1];          /* if printing, the name of the printer.
                                           Otherwise, "-". */
        time_t o_date;                  /* date of entry into output queue */
        short o_flags;                  /* See below for flag values */
};

/* Value interpretation for o_flags: */

#define O_DEL       1                   /* Request deleted */
#define O_PRINT     2                   /* Request now printing */
```

Figure 2. Outputq Entry

4.5 Pstatus

The binary file **pstatus** contains one entry of status information for each printer. Printer status entries are detailed in Figure 3. Entries are added and removed by the *lpadmin* command and are modified by the *cancel*, *enable*, *disable* and *lpsched* commands.

```
struct pstat {                          /* printer status entry */
        char p_dest[DESTMAX+1];         /* name of printer */
        int p_pid;                      /* if busy, process id that is printing, otherwise 0 */
        char p_rdest[DESTMAX+1];        /* if busy, the destination designated
                                           by the user to lp, otherwise "-" */
        int p_seqno;                    /* if busy, seq # of printing request */
        time_t p_date;                  /* date last enabled or disabled */
        char p_reason[P_RSIZE];         /* if enabled, then "enabled", otherwise
                                           the reason the printer has been disabled. */
        short p_flags;                  /* See below for flag values */
};

#define P_ENAB      1                   /* printer enabled */
#define P_AUTO      2                   /* disable printer automatically */
#define P_BUSY      4                   /* printer now printing a request */
```

Figure 3. Pstatus Entry

4.6 Qstatus

The binary file **qstatus** contains one entry per destination which tells if the *lp* command is accepting requests. **Qstatus** entries have the form shown in Figure 4 and are added and removed by the *lpadmin* command and modified by the *accept*, *reject* and *lpmove* commands.

```
struct qstat {                          /* queue status entry */
        char q_dest[DESTMAX+1];         /* destination */
        short q_accept;                 /* TRUE iff lp accepting requests for dest,
                                           otherwise FALSE.*/
        time_t q_date;                  /* date status last modified */
        char q_reason[Q_RSIZE];         /* if accepting then "accepting",
                                           otherwise the reason requests for dest are
                                           being rejected by lp */
};
```

Figure 4. Qstatus Entry

4.7 Seqfile

The file **seqfile** contains the sequence number (terminated by a new-line) of the last request id that was assigned by the *lp* command. This number is incremented by *lp* for each request. When it reaches a maximum (defined in **lp.h**) it is reset to 1. If this file is missing then *lp* will create a new file containing the number 1.

4.8 Lock Files

Several lock files are maintained in order to guarantee LP commands exclusive access to data files. They are binary files which contain the process id of the locking process. A list of lock files and their associated data files follows:

Lock File	*Data File*
OUTQLOCK	outputq
PSTATLOCK	pstatus
QSTATLOCK	qstatus
SEQLOCK	seqfile

Lock files "expire" after a given time interval and may be unlinked by any LP process. Thus, commands that lock a data file for longer than this interval must update the modification time on the lock file. The creation, updating and unlinking of lock files is handled automatically by the LP low level file access routines.

Another lock file, **SCHEDLOCK**, is present while *lpsched* is running to ensure that only one invocation of *lpsched* is active. Unlike other lock files, **SCHEDLOCK** has no expiration time.

Caution: any processes that need to concurrently lock more than one lock file should lock them in the following order to avoid deadlock:

OUTQLOCK, PSTATLOCK, QSTATLOCK, SEQLOCK

Failure to release a lock file may also cause deadlock.

4.9 Class

The **class** directory contains one file per LP class which lists the members of the class, one per line. The name of the file is the same as the class name. Each class member is an LP printer and may not be an LP class. Every class must always have at least one member. Class files are created, modified and deleted by the *lpadmin* command.

4.10 Interface

The **interface** directory contains one executable program per printer with the same name as the printer. When *lpsched* chooses an output request, **dest−seqno**, that was requested by user **logname**, to be printed on printer **pr**, it invokes interface program **pr** in the following way:

pr dest−seqno logname title copies options file ...

where
copies is the number of copies requested
title is the optional title supplied to *lp* or null
options is a blank-separated string of options requested by the user to *lp* or null
file is the full path name of a file to be printed

The interface program is invoked with its standard output and standard error output directed to the printer's device. If file access modes permit, the device is opened for reading and writing. The interface's standard input is taken from **/dev/null**. Interface programs may be shell procedures or compiled C programs. They may be supplied by an LP Administrator or selected from a set of model interface programs (see *Model* below). Interface programs are supplied by LP Administrators via the *lpadmin* command.

4.11 Member

The **member** directory contains one file per LP printer with the same name as the printer. The first line of the file is the full path name of the device associated with the printer. Following lines (if any) are the names of classes to which the printer belongs. A printer need not belong to any classes and may belong to more than one. Member files are created, modified and removed by the *lpadmin* command.

4.12 Model

The **model** directory contains several printer interface programs that are distributed with the LP system. The names of these files bear no relationship to LP printers and class names. Copies of these programs may be customized by an LP Administrator to be used as printer interface programs. No new model interfaces can be added to the system.

4.13 Request

The **request** directory contains one directory for each LP destination with the same name as the destination. Each destination's request subdirectory holds information pertaining to pending requests for that destination.

Each request subdirectory contains request files and data files. These files are created by the *lp* command to pass information to *lpsched* and are deleted by the *cancel*, *disable* and *lpsched* commands, and may be moved by the *lpmove* command.

The name of the request file for output request **dest−seqno** is **r−seqno**. It has entries of the form:

flag value

where *flag* is a single character in column 1, column 2 is blank and an optional *value* starts in column 3. Legal flags, as defined mnemonically in **lp.h**, are summarized in Table 2. The order of entries in a request file is the same order that they are listed in Table 2. The R_TITLE, R_COPIES, R_OPTIONS and one or more R_FILE entries are mandatory.

TABLE 2. Request File Entries

FLAG	*VALUE*
R_TITLE	Optional title supplied to *lp* or null
R_COPIES	Number of copies requested
R_OPTIONS	Printer- and Class-dependent options separated by white space
R_FILE	Name of data file to be printed; any file name not beginning with "/" is assumed to be in the request subdirectory along with the request file
R_MAIL	Logname of person to send mail to after request has been printed
R_WRITE	Logname of person to write to after request has been printed

A request file is associated with zero or more data files. Data files for request **dest−seqno** have the name **dn−seqno**, where **n** is a non-negative integer. These files contain data to be printed.

Examples:

1. $ pr file | lp
   request id is x−50 (standard input)

   The directory **request/x** will contain the request file **r−50** and the data file **d0−50** which is a copy of the standard input to *lp*. File **r−50**:

   R_TITLE
   R_COPIES 1
   R_OPTIONS
   R_FILE d0−50

2. $ lp −c file1 file2
   request id is x−51 (2 files)

   The **−c** option causes *lp* to copy files before returning to the user. The directory **request/x** will contain the request file **r−51** and the data files **d0−51** (a copy of file1) and **d1−51** (a copy of file2). File **r−51**:

   R_TITLE
   R_COPIES 1
   R_OPTIONS
   R_FILE d0−51
   R_FILE d1−51

3. $ lp file
   request id is x−52 (1 file)

   The directory **request/x** will contain the request file **r−52**. If *file* can be linked to this directory it will be named d0−52. In this case, file **r−52** contains:

   R_TITLE
   R_COPIES 1
   R_OPTIONS
   R_FILE d0−52

   On the other hand, if *file* can't be linked, no data file is created and file **r−52** contains:

   R_TITLE
   R_COPIES 1
   R_OPTIONS
   R_FILE fullfile

   Fullfile is the full path name of *file*.

5. LP SCHEDULER INTERNALS

5.1 Overview

The LP scheduler, *lpsched*, services requests in the output queue, **outputq**, first come first served, invoking the appropriate interface program to print each request. It is the only demon in the LP system and runs continuously unless it is stopped by the *lpshut* command or the computer system is stopped. It is present even when there are no pending output requests, in which case it sleeps awaiting a message on FIFO.

5.2 Interaction With Other LP Commands

Because it would be inefficient for *lpsched* to perform file I/O each time it needed to know the relationships between printers, classes, requests and devices, *lpsched* maintains its own structures which provide this information more easily. This burdens LP commands by requiring them to inform *lpsched* (on FIFO) of changes to LP data in addition to updating the data files. The former step is required in order to keep the file structure consistent with *lpsched*'s in-memory data. It is this duplication of information that allows LP commands to be used even if *lpsched* is not running.

As an example, let us consider how the *lp* command works. When a request is made to *lp* it builds the request and data files, locks **OUTQLOCK**, adds the new request entry to **outputq**, writes an F_REQUEST message to *lpsched* on FIFO which describes the new request and then unlocks **OUTQLOCK**. The time during which **OUTQLOCK** is locked is a non-interruptible critical section, so signals are ignored. Most LP commands follow this pattern of:

1. lock one or more lock files
2. modify one or more data files
3. send a message to *lpsched* on FIFO
4. unlock the lock files

5.3 Data Structures

When *lpsched* is started it internalizes the information in the LP data files in an in-memory network of circular double-linked lists. Subsequent messages read from FIFO cause this network to be updated so that the lists are kept consistent with the files. The main component of *lpsched*'s lists is the **dest** node shown in Figure 5. There is one of these structures for each destination giving its name and type (class or printer). Nodes that are printers also indicate status (busy or idle, enabled or disabled) as well as information concerning the currently printing request.

```
struct dest {                           /* destination node */
        char *d_dname;                  /* name of destination */
        int d_status;                   /* status of destination -- see below */
        char *d_device;                 /* full path name of device for printer */
        int d_pid;                      /* process id of busy printer */
        struct outlist *d_print;        /* output request currently printing */
        struct dest *d_dnext;           /* next destination */
        struct dest *d_dprev;           /* previous destination */
        struct dest *d_tnext;           /* next destination of same type */
        struct dest *d_tprev;           /* previous destination of same type */
        struct destlist *d_class;       /* class list for printers, member list for classes */
        struct outlist *d_output;       /* list of output requests for dest */
};

/* The following flags are used to interpret dest.d_status */

#define D_PRINTER  1                    /* destination is a printer */
#define D_CLASS    2                    /* destination is a class */
#define D_ENABLED  8                    /* printer is active */
#define D_BUSY     16                   /* printer is busy */
```

Figure 5. Destination Node

Three global **dest** nodes serve as list heads to ease the traversal of the network:

dest	links all destinations in the d_dnext and d_dprev fields
printer	links all printers in the d_tnext and d_tprev fields
class	links all classes in the d_tnext and d_tprev fields

Each printer node contains a linked list of destinations indicating which classes it belongs to. Class nodes have lists of the same format showing which printers are members. Destination lists, as shown in Figure 6, point to **dest** nodes.

```
struct destlist {                       /* destination list node */
        struct dest *dl_dest;           /* pointer to destination */
        struct destlist *dl_next;       /* pointer to next destination */
        struct destlist *dl_prev;       /* pointer to previous destination in list */
};
```

Figure 6. Destination List Node

Because output may be directed to classes or printers, every destination has an associated output request list. Each list is ordered according to the time the F_REQUEST messages were received by *lpsched* from *lp*. The format of output request lists is shown in Figure 7.

```
struct outlist {                        /* output request list node */
        int ol_seqno;                   /* sequence number assigned by lp */
        char *ol_name;                  /* logname of requester */
        int ol_time;                    /* time request was received by lpsched */
        struct dest *ol_dest;           /* pointer to request destination */
        struct dest *ol_print;          /* if printing, a pointer to the printer */
        struct outlist *ol_next;        /* next output request in list */
        struct outlist *ol_prev;        /* previous output request in list */
};
```

Figure 7. Output Request List Node

5.4 Printing a Request

Lpsched is ready to print a request when one of the following messages is received and when one of that message's associated conditions is met:

Message	*Conditions*
F_REQUEST dest seqno user	1. dest is an enabled, idle printer 2. dest is a class which contains an enabled, idle printer
F_MORE pr	1. there is a pending request queued for pr 2. there is a pending request queued for a class which pr belongs to
F_ENABLE pr	1. there is a pending request queued for pr 2. there is a pending request queued for a class which pr belongs to
F_DISABLE pr	pr is busy and the request it is currently printing is queued for a class which contains pr and a member of that class is enabled and idle

When a request is ready to be printed, *lpsched* forks so that its child may do the printing and the parent can continue scheduling other requests. It is the child that executes the interface program and waits for its completion. A non-zero exit status indicates that the interface encountered errors while printing the request. If errors occurred or if the user requested notification of the completion of the printing, mail is sent or a message is written to the requester's terminal. The **outputq** entry is deleted and the request and data files are removed. The parent is informed that the printer is ready to print another request via an F_MORE message on FIFO and the child exits.

Several processes are concurrently active during the printing of a request. Because UNIX systems are typically configured to impose limits on the number of concurrently active processes

per user id (except for root) and because most LP programs must be owned by user lp and because they require set-user-id permission, the number of printers that LP can support is affected. Unless the LP system is owned by root (this is not encouraged) there is a limit on the number of LP printers that may be printing simultaneously. The number of active processes per print request includes the child of the scheduler and the interface program and any of its children. Model interface programs, for example, are shell procedures that usually have the form:

```
commands
(
    commands
) | filter
exit
```

Each request that is printed by a model interface creates two invocations of the shell, an invocation of a device filter and a process to execute a command within the parentheses in addition to a child of the scheduler. With a limit of 25 active processes per user, for example, an LP system with an LP administrator other than root would be able to support up to four line printers using typical model interface programs.

5.5 Cancellation of Requests that are Partially Printed

When a partially printed request is canceled via the *cancel* or *disable* command, the process id found in the **pstatus** entry is signaled with **SIGTERM**. This is the process id of the interface program itself, not the immediate child of the scheduler. It is up to the interface to clean up and then exit. The child of the scheduler waits for the death of the interface and exits. The scheduler then waits for the death of its child.

When *lpsched* is stopped by the *lpshut* command or when it is signaled with **SIGTERM**, it broadcasts this signal to all of its children, which, in turn, terminate the execution of the interface programs. *Lpsched* then removes **SCHEDLOCK** and exits. When printing is terminated this way or by disabling a busy printer (without canceling the request), the requests that were aborted will be reprinted in their entirety.

6. USING LP AS A GENERAL PURPOSE SPOOLER

Although the documentation and commands refer to LP as a line printer spooling system, it was designed with general purpose spooling in mind. Several features allow LP to be customized for a variety of spooling applications. The *lp* command never makes any assumptions about the kind of files it is supplied. No pagination is added and no checks are made for non-ascii input. Thus, *lp* can pass command files, nroff/troff input files, binary data files, executable files, ascii text files, etc. to arbitrary interface programs. *Lp* also allows users to pass options from the *lp* command line to interface programs using the **−o** key letter. It is up to the LP Administrator to supply interface programs to perform the desired functions on input files. The devices associated with printers need not be line printers. They must be writable by LP and may be any type of file (even **/dev/null**). By designing interface programs and by placing new interpretations on destinations and devices LP can perform many diverse functions.

Example:

Many installations use special purpose software to batch *nroff* requests so that they can limit the number of concurrently executing *nroff* commands. LP can be used for batch processing of this and other commands that place a heavy load on the system. Each "printer" can be thought of as a command processor. Input files (built by a front-end interface to *lp*) are shell procedures which contain *nroff* command lines and environment information. The role of the interface program is to execute the nroff command in the user's environment as of the time *lp* was invoked. The output may be directed to a file or a printer designated by the user. The devices associated with the processors could be log files or line printers. By grouping *n* of these

processors in a class, users are limited to *n* concurrent executions of frequently used, heavy load commands. Furthermore, queuing to these destinations and the running of the command processors are under the control of the *accept*, *reject*, *enable* and *disable* commands.

7. EXTENSIONS

It is hoped that any future enhancements to LP will not take away from its generality. It would have been easy, for instance, to add dozens of printer-specific options to the *lp* command. This was not done because LP makes it easy for LP Administrators to add these options in printer interface programs and for users to take advantage of them by sliding them past *lp*. On the other hand, there are enhancements that could make LP even more useful while retaining its generality.

Lpsched services requests on a first come first served basis. This may be undesirable when there are a limited number of printers and it is desirable to schedule small print jobs ahead of larger ones. Care must be taken to avoid penalizing the larger requests too severely. *Lpsched* could be enhanced to enforce such a scheduling discipline. User-assigned priorities could be added to the *lp* command in order to affect *lpsched*'s scheduling algorithm. Another useful feature is to allow users to inhibit requests from printing while leaving them queued. Subsequently, the held requests could be released or canceled. Another enhancement to LP would allow different queues to build for the same destination in order to implement the idea of "peak period" or "overnight" queues.

The *lpadmin* command requires that *lpsched* must not be running before it is going to attempt to alter the LP configuration. This restriction was imposed to simplify the initial version of *lpsched*. In cases where a configuration is frequently undergoing changes it is a nuisance to have to shut the scheduler before using *lpadmin*. Shutting the scheduler, of course, means that all printing stops.

The above features were not considered absolutely essential and would have greatly increased the complexity of the initial version of the package. The author believes that it would not require a major effort to add these new capabilities to LP. The design would not need to be radically changed to introduce these enhancements.

8. SUMMARY

To the best of the author's knowledge, LP is the only centrally supported spooler under UNIX which offers all of the following features in a single package:

- Printers may be grouped into classes.
- Each printer may belong to several or no classes.
- The spooler may be reconfigured to meet the needs of specific users.
- The spooling function is separated from the printing function. Any device or writable file may be spooled to by a user-supplied interface program.
- LP can be used for off-line printing as well as for other spooling functions.

REFERENCE

[1] Kliegman, J. R. *LP Administrator's Guide*, Bell Laboratories.

LP Administrator's Guide

J. R. Kliegman

Bell Laboratories
Piscataway, New Jersey 08854

1. INTRODUCTION

LP is a system of commands that performs diverse spooling functions under the UNIX† operating system. Because its primary application is off-line printing, this paper focuses mainly on spooling to line printers. LP allows administrators to customize the system to spool to a collection of line printers of any type and to group printers into logical classes in order to maximize the throughput of the devices. Users are provided the capabilities of queuing and canceling print requests, preventing and allowing queuing to and printing on devices, starting and stopping LP from processing requests, changing their configuration of printers and finding the status of the LP system. This memo describes the role of an LP Administrator (LPA) in performing restricted functions and overseeing the smooth operation of LP.

The remainder of this paper is organized as follows: Section 2 presents an overview of the features of LP and defines terms that will be used throughout the memo. See [1] for a detailed description of the implementation of LP. Section 3 tells how to build an LP system. Sections 4-11 describe how to perform administrative functions using LP commands. Section 12 covers how to write printer interface programs, Section 13 indicates how to set up hardwired printers and login terminals to be used with LP and the final section summarizes the role of the LPA.

2. OVERVIEW OF LP FEATURES

2.1 Definitions

We will define several terms before presenting a brief summary of LP commands. LP was designed with the flexibility to meet the needs of users on different UNIX systems. Changes to LP's configuration (see below) are performed by the *lpadmin*(1M) command. (A parenthesized number immediately following a command name refers to that section of the *UNIX User's Manual.*)

LP makes a distinction between printers and printing devices. A *device* is a physical peripheral device or a file and is represented by a full UNIX path name. A *printer* is a logical name that represents a device. At different points in time, a printer may be associated with different devices. A *class* is a name given to an ordered list of printers. Every class must contain at least one printer. Each printer may be a member of zero or more classes. A *destination* is a printer or a class. One destination may be designated as the *system default destination*. The *lp*(1) command will direct all output to this destination unless the user specifies otherwise. Output that is routed to a printer will be printed only by that printer, whereas output directed to a class will be printed by the first available class member.

Each invocation of *lp* creates an output *request* that consists of the files to be printed and options from the *lp* command line. An *interface program* which formats requests must be supplied for each printer. The LP scheduler, *lpsched*(1M), services requests for all destinations by routing requests to interface programs to do the printing on devices. An LP *configuration* for a system consists of devices, destinations and interface programs.

---

† UNIX is a trademark of Bell Laboratories.

2.2 Commands

2.2.1 Commands for General Use

Lp(1) is used to request the printing of files. It creates an output request and returns a *request id* of the form:

dest−seqno

to the user, where *seqno* is a unique sequence number across the entire LP system and *dest* is the destination where the request was routed.

Cancel is used to cancel output requests. The user supplies request ids as returned by *lp* or printer names, in which case the currently printing requests on those printers are canceled.

Disable prevents *lpsched* from routing output requests to printers.

Enable(1) allows *lpsched* to route output requests to printers.

2.2.2 Commands for LP Administrators

Each LP system must designate a person or persons as LP Administrator to perform the restricted functions listed below. Either the super-user or any user who is logged into UNIX as **lp** qualifies as an LP Administrator. All LP files and commands are owned by lp, except for *lpadmin* and *lpsched*, which are owned by root. The following commands will be described in more detail later in this memo.

Lpadmin(1M) modifies the LP configuration. Many features of this command cannot be used when *lpsched* is running.

Lpsched(1M) routes output requests to interface programs which do the printing on devices.

Lpshut stops *lpsched* from running. All printing activity is halted, but the other LP commands may still be used.

Accept(1M) allows *lp* to accept output requests for destinations.

Reject prevents *lp* from accepting requests for destinations.

Lpmove moves output requests from one destination to another. Whole destinations may be moved at once. This command cannot be used when *lpsched* is running.

3. BUILDING LP

All LP commands are built from source code that resides in the **/usr/src/cmd/lp** directory including the make file, **lp.mk**. Unless some of the definitions in **lp.mk** are changed, LP may be installed only by the super-user. Before installing a new LP system, make sure there is a login called **lp** on your system and that the spool directory, **/usr/spool/lp**, does not exist. To install LP, perform the following:

```
cd /usr/src/cmd/lp
make −f lp.mk install
```

This builds all LP commands and creates an initial LP configuration consisting of no printers, classes or default destination. LP must be configured by an LPA using the *lpadmin* command in order to create a useful spooler.

In addition, add the following code to /etc/rc:

```
rm −f /usr/spool/lp/SCHEDLOCK
/usr/lib/lpsched
echo "LP scheduler started"
```

This starts the LP scheduler each time that UNIX is restarted.

Several variables in **lp.mk** may be changed before installing LP to customize the system:

Variable	*Default Value*	*Meaning*
SPOOL	/usr/spool/lp	spool directory
ADMIN	lp	logname of LP Administrator
GROUP	bin	group that owns LP commands and data
ADMDIR	/usr/lib	administrator commands reside here
USRDIR	/usr/bin	user commands reside here

If an existing LP spool directory is corrupted (but not the LP programs) or if it needs to be rebuilt from scratch, make sure that *lpsched* is not running and perform the following as superuser:

1. Make copies of any interface programs that are not standard LP software. DO NOT make these copies underneath the spool directory. The path name for printer p is **/usr/spool/lp/interface/p**.
2. rm −fr /usr/spool/lp
3. make −f lp.mk new (this recreates the bare LP configuration described above).

WARNINGS:

1. Some LP commands invoke other LP commands. Moving them after they are built will cause some commands to fail.
2. The files under the SPOOL directory should be modified *only by LP commands*.
3. All LP commands require set-user-id permission. If this is removed, the commands will fail.

4. CONFIGURING LP − THE LPADMIN COMMAND

Changes to the LP configuration should be made by using the *lpadmin* command and not by hand. *Lpadmin* will not attempt to alter the LP configuration when *lpsched* is running, except where explicitly noted below.

4.1 Introducing New Destinations

The following information must be supplied to *lpadmin* when introducing a new printer:

1. The printer name (−**p**printer) is an arbitrary name which must conform to the following rules:
   - It must be no longer than fourteen characters.
   - It must consist solely of alphanumeric characters and underscores.
   - It must not be the name of an existing LP destination (printer or class).
2. The device associated with the printer (−**v**device). This is the path name of a hardwired printer, a login terminal, or other file that is writable by lp.
3. The printer interface program. This may be specified in one of three ways:
   - It may be selected from a list of model interfaces supplied with LP (−**m**model).
   - It may be the same interface that an existing printer uses (−eprinter).
   - It may be a program supplied by the LPA (−**i**interface).

Information that need not always be supplied when creating a new printer includes:

1. The user may specify **−h** to indicate that the device for the printer is hardwired or the device is the name of a file (this is assumed by default). If, on the other hand, the device is the path name of a login terminal, then **−l** must be included on the command line. This indicates to *lpsched* that it must automatically disable this printer each time *lpsched* starts running. This fact is reported by *lpstat* when it indicates printer status:

   ```
   $ lpstat −pa
   printer a (login terminal) disabled since Oct 31 11:15 −
           disabled by scheduler: login terminal
   ```

   This is done because device names for login terminals can be (and usually are) associated with different physical devices from day to day. If the scheduler did not take this action, somebody might log in and be surprised that LP is spooling to his/her terminal!

2. The new printer may be added to an existing class or added to a new class (−cclass). New class names must conform to the same rules for new printer names.

Examples:

The following examples will be referenced by further examples in later sections:

1. Create a printer called pr1 whose device is /dev/printer and whose interface program is the model hp interface:

   ```
   $ /usr/lib/lpadmin −ppr1 −v/dev/printer −mhp
   ```

2. Add a printer called pr2 whose device is /dev/tty22 and whose interface is a variation of the model prx interface. It is also a login terminal:

   ```
   $ cp /usr/spool/lp/model/prx xxx
      < edit xxx here >
   $ /usr/lib/lpadmin −ppr2 −v/dev/tty22 −ixxx −l
   ```

3. Create a printer called pr3 whose device is /dev/tty23. pr3 will be added to a new class called cl1 and will use the same interface as printer pr2:

   ```
   $ /usr/lib/lpadmin −ppr3 −v/dev/tty23 −epr2 −ccl1
   ```

4.2 Modifying Existing Destinations

Modifications to existing destinations must always be made with respect to a printer name (−pprinter). The modifications may be one or more of the following:

1. The device for the printer may be changed (−vdevice). If this is the only modification, than this may be done even while *lpsched* is running. This facilitates changing devices for login terminals.
2. The printer interface program may be changed (−**m**model, −eprinter, −iinterface).
3. The printer may be specified as hardwired (−**h**) or as a login terminal (−**l**).
4. The printer may be added to a new or existing class (−cclass).
5. The printer may be removed from an existing class (−rclass). Removing the last remaining member of a class causes the class to be deleted. No destination may be removed if it has pending requests. In that case, *lpmove* or *cancel* should be used to move or delete the pending requests.

Examples:

These examples are based on the LP configuration created by those in the previous section.

1. Add printer pr2 to class cl1:

   ```
   $ /usr/lib/lpadmin -ppr2 -ccl1
   ```

2. Change pr2's interface program to the model prx interface, change its device to /dev/tty24, and add it to a new class called cl2:

   ```
   $ /usr/lib/lpadmin -ppr2 -mprx -v/dev/tty24 -ccl2
   ```

   Note that printers pr2 and pr3 now use different interface programs even though pr3 was originally created with the same interface as pr2. Printer pr2 is now a member of two classes.

3. Specify printer pr2 as a hardwired printer:

   ```
   $ /usr/lib/lpadmin -ppr2 -h
   ```

4. Add printer pr1 to class cl2:

   ```
   $ /usr/lib/lpadmin -ppr1 -ccl2
   ```

   The members of class cl2 are now pr2 and pr1, in that order. Requests routed to class cl2 will be serviced by pr2 if both pr2 and pr1 are ready to print, otherwise they will be printed by the one which is next ready to print.

5. Remove printers pr2 and pr3 from class cl1:

   ```
   $ /usr/lib/lpadmin -ppr2 -rcl1
   $ /usr/lib/lpadmin -ppr3 -rcl1
   ```

   Because pr3 was the last remaining member of class cl1, the class is removed.

6. Add pr3 to a new class called cl3:

   ```
   $ /usr/lib/lpadmin -ppr3 -ccl3
   ```

4.3 Specifying the System Default Destination

The system default destination may be changed even when *lpsched* is running.

Examples:

1. Establish class cl1 as the system default destination:

   ```
   $ /usr/lib/lpadmin -dcl1
   ```

2. Establish no default destination:

   ```
   $ /usr/lib/lpadmin -d
   ```

4.4 Removing Destinations

Classes and printers may be removed only if there are no pending requests that were routed to them. Pending requests must either be canceled using *cancel* or moved to other destinations using *lpmove* before destinations may be removed. If the removed destination is the system default destination, then the system will have no default destination until it respecified. When the last remaining member of a class is removed, then the class is also removed. The removal of a class never implies the removal of printers.

Examples:

1. Make printer pr1 the system default destination:

   $ /usr/lib/lpadmin −dpr1

   Remove printer pr1:

   $ /usr/lib/lpadmin −xpr1

   Now there is no system default destination.

2. Remove printer pr2:

   $ /usr/lib/lpadmin −xpr2

   Class cl2 is also removed, because pr2 was its only member.

3. Remove class cl3:

   $ /usr/lib/lpadmin −xcl3

   Class cl3 is removed, but printer pr3 remains.

5. MAKING AN OUTPUT REQUEST − THE LP COMMAND

Once LP destinations have been created, users may request output by using the *lp* command. The request id that is returned may be used to see if the request has been printed or to cancel the request.

Lp determines the destination of a request by checking the following list in order:

- If the user specifies −d*dest* on the command line, then the request is routed to *dest*.
- If the environment variable **LPDEST** is set, the request is routed to the value of *LPDEST*.
- If there is a system default destination, then the request is routed there.
- Otherwise, the request is rejected.

Examples:

1. There are at least four ways to print the password file on the system default destination:

   ```
   lp /etc/passwd
   lp < /etc/passwd
   cat /etc/passwd | lp
   lp −c /etc/passwd
   ```

   The last three ways cause copies of the file to be printed, whereas the first way prints the file directly. Thus, if the file is modified between the time the request is made and the time it is actually printed, then the changes will be reflected in the output.

2. Print two copies of file abc on printer xyz and title the output "my file":

   ```
   pr abc | lp −dxyz −n2 −t"my file"
   ```

3. Print file xxx on a Diablo 1640 printer called zoo in 12-pitch and write to the user's terminal when printing has completed:

   ```
   lp −dzoo −o12 −w xxx
   ```

   In this example, **12** is an option that is meaningful to the model Diablo 1640 interface program that prints output in 12-pitch mode (see *lpadmin*(1M)).

6. FINDING LP STATUS – LPSTAT

The *lpstat* command is used to find status information about LP requests, destinations and the scheduler.

Examples:

1. List the status of all pending output requests made by this user:

   lpstat

   The status information for a request includes the request id, the logname of the user, the total number of characters to be printed and the date and time the request was made.

2. List the status of printers p1 and p2:

   lpstat –pp1,p2

7. CANCELING REQUESTS – CANCEL

LP requests may be canceled using the *cancel* command. Two kinds of arguments may be given to the command – request ids and printer names. The requests named by the request ids are canceled and requests that are currently printing on the named printers are canceled. Both types of arguments may be intermixed.

Example:

Cancel the request that is now printing on printer xyz:

```
cancel xyz
```

If the user that is canceling a request is not the same one that made the request, then mail is sent to the owner of the request. LP allows *any* user to cancel requests in order to eliminate the need to find LP Administrators when unwanted output is be purged.

8. ALLOWING AND REFUSING REQUESTS – ACCEPT AND REJECT

When a new destination is created, *lp* will reject requests that are routed to it. When the LP Administrator is sure that it is set up correctly he or she should allow *lp* to accept requests for that destination. The *accept* command performs this function.

Sometimes it is necessary to prevent *lp* from routing requests to destinations. If printers have been removed or are waiting to be repaired or if too many requests are building for printers then it may be desirable to cause *lp* to reject requests for those destinations. The *reject* command performs this function. After the condition that led to the rejection of requests has been remedied, the *accept* command should be used to allow requests to be taken again.

The acceptance status of destinations is reported by the **–a** option of *lpstat*.

Examples:

1. Cause *lp* to reject requests for destination xyz:

   /usr/lib/reject –r"printer xyz in need of repair" xyz

   Any users that try to route requests to xyz will encounter the following:

   ```
   $ lp –dxyz file
   lp: can't accept requests for destination "xyz" –
           printer xyz in need of repair
   ```

2. Allow *lp* to accept requests routed to destination xyz:

   /usr/lib/accept xyz

9. ALLOWING AND INHIBITING PRINTING – ENABLE AND DISABLE

The *enable* command allows the LP scheduler to print requests on printers. That is, the scheduler routes requests only to the interface programs of enabled printers. Note that it is possible to enable a printer but to prevent further requests from being routed to it.

The *disable* command undoes the effects of the *enable* command. It prevents the scheduler from routing requests to printers, independently of whether or not *lp* is allowing them to accept requests. Printers may be disabled for several reasons including malfunctioning hardware, paper jams and end of day shutdowns. If a printer is busy at the time it is disabled, then the request that it was printing will be reprinted in its entirety either on another printer (if the request was originally routed to a class of printers) or on the same one when the printer is re-enabled. The –c option causes the currently printing requests on busy printers to be canceled in addition to disabling the printers. This is useful if strange output is causing a printer to behave abnormally.

Example:

Disable printer xyz because of a paper jam:

```
$ disable –r"paper jam" xyz
printer "xyz" now disabled
```

Find the status of printer xyz:

```
$ lpstat –pxyz
printer "xyz" disabled since Jan 5 10:15 –
        paper jam
```

Now, re-enable xyz:

```
$ enable xyz
printer "xyz" now enabled
```

10. MOVING REQUESTS BETWEEN DESTINATIONS – LPMOVE

Occasionally, it is useful for LP Administrators to move output requests between destinations. For instance, when a printer is down for repairs it may be desirable to move all of its pending requests to a working printer. This is one way to use the *lpmove* command. The other use of this command is to move specific requests to a different destination. *Lpmove* will refuse to move requests while the LP scheduler is running.

Examples:

1. Move all requests for printer abc to printer xyz:

   ```
   $ /usr/lib/lpmove abc xyz
   ```

   All of the moved requests are renamed from abc–nnn to xyz–nnn. As a side effect, destination abc is no longer accepting further requests.

2. Move requests zoo–543 and abc–1200 to printer xyz:

   ```
   $ /usr/lib/lpmove zoo–543 abc–1200 xyz
   ```

   The two requests are now renamed xyz–543 and xyz–1200.

11. STOPPING AND STARTING THE SCHEDULER – LPSHUT AND LPSCHED

Lpsched is the program that routes the output requests that were made with *lp* through the appropriate printer interface programs to be printed on line printers. Each time the scheduler routes a request to an interface program, it records an entry in the log file, **/usr/spool/lp/log**. This entry contains the logname of the user who made the request, the request id, the name of

the printer that the request is being printed on and the date and time that printing first started. In the case that a request has been restarted, more than one entry in the log file may refer to the request. The scheduler also records error messages in the log file. When *lpsched* is started, it renames **/usr/spool/lp/log** to **/usr/spool/lp/oldlog** and starts a new log file.

No printing will be performed by the LP system unless *lpsched* is running. Use the command:

```
lpstat −r
```

to find the status of the LP scheduler.

Lpsched is normally started by the **/etc/rc** program as described above and continues to run until UNIX is shut down. The scheduler operates in the **/usr/spool/lp** directory. When it starts running, it will exit immediately if a file called **SCHEDLOCK** exists. Otherwise, it creates this file in order to prevent more than one scheduler from running at the same time.

Occasionally, it is necessary to shut the scheduler in order to reconfigure LP or to rebuild the LP software. The command

```
/usr/lib/lpshut
```

causes *lpsched* to stop running and terminates all printing activity. All requests that were in the middle of printing will be reprinted in their entirety when the scheduler is restarted.

To restart the LP scheduler, use the command

```
/usr/lib/lpsched
```

Shortly after this command is entered, *lpstat* should report that the scheduler is running. If not, it is possible that a previous invocation of *lpsched* exited without removing **SCHEDLOCK**, so try the following:

```
rm −f /usr/spool/lp/SCHEDLOCK
/usr/lib/lpsched
```

The scheduler should be running now.

12. PRINTER INTERFACE PROGRAMS

Every LP printer must have an interface program which does the actual printing on the device that is currently associated with the printer. Interface programs may be shell procedures, C programs, or any other executable programs. LP's model interfaces are all written as shell procedures and can be found in the **/usr/spool/lp/model** directory. At the time *lpsched* routes an output request to a printer P, the interface program for P is invoked in the directory **/usr/spool/lp** as follows:

```
interface/P id user title copies options file ...
```

where

id is the request id returned by *lp*
user is the logname of the user who made the request
title is the optional title specified by the user
copies is the number of copies requested by the user
options is a blank-separated list of class- or printer-dependent options specified by the user
file is the full path name of a file to be printed

Examples:

The following examples are requests made by user "smith" with a system default destination of printer "xyz". Each example lists an *lp* command line, followed by the corresponding command line generated for printer xyz's interface program:

1. lp /etc/passwd /etc/group
   interface/xyz xyz−52 smith "" 1 "" /etc/passwd /etc/group
2. pr /etc/passwd | lp −t"users" −n5
   interface/xyz xyz−53 smith users 5 "" /usr/spool/lp/request/xyz/d0−53
3. lp /etc/passwd −oa −ob
   interface/xyz xyz−54 smith "" 1 "a b" /etc/passwd

When the interface program is invoked, its standard input comes from /dev/null and both the standard output and standard error output are directed to the printer's device. Devices are opened for reading as well as writing when file modes permit. In the case where a device is a regular file, all output is appended to the end of the file.

Given the command line arguments and the output directed to a device, interface programs may format their output in any way they choose. Interface programs must ensure that the proper stty modes (terminal characteristics such as baud rate, output options, etc.) are in effect on the output device. This may be done as follows in a shell interface only if the device is opened for reading:

```
stty mode ... <&1
```

That is, take the standard input for the stty command from the device.

When printing has completed, it is the responsibility of the interface program to exit with a code indicative of the success of the print job. Exit codes are interpreted by *lpsched* as follows:

CODE	*MEANING TO LPSCHED*
zero	The print job has completed successfully.
1 to 127	A problem was encountered in printing this particular request (e.g., too many non-printable characters). This problem won't affect future print jobs. *Lpsched* notifies users by mail that there was an error in printing the request.
greater than 127	These codes are reserved for internal use by *lpsched*. Interface programs must not exit with codes in this range.

When problems that are likely to affect future print jobs occur (e.g., a device filter program is missing), the interface programs would be wise to disable printers so that print requests are not lost. When a busy printer is disabled, the interface program will be terminated with signal 15.

13. SETTING UP HARDWIRED DEVICES AND LOGIN TERMINALS AS LP PRINTERS

13.1 Hardwired Devices

As an example of how to set up a hardwired device for use as an LP printer, let us consider using tty line 15 as printer xyz. As super-user, perform the following:

1. Avoid unwanted output from non-LP processes and ensure that LP can write to the device:

   ```
   $ chown lp /dev/tty15
   $ chmod 600 /dev/tty15
   ```

2. Change /etc/inittab so that tty15 is not a login terminal. In other words, ensure that /etc/getty is not trying to log users in at this terminal. Change the entries for line 15 to:

   ```
   1:15:o:
   2:15:o:
   ```

   Enter the command:

   ```
   $ init 2
   ```

   If there is currently an invocation of /etc/getty running on tty15, then kill it. Now, and when UNIX is rebooted, tty15 will be initialized with default stty modes. Thus, it is up to LP interface programs to establish the proper baud rate and other stty modes for correct printing to occur.

3. As explained above in Section 4.1, introduce printer xyz to LP using the model Printronix interface program:

   ```
   $ /usr/lib/lpadmin −pxyz −v/dev/tty15 −mprx
   ```

4. When xyz is created, it will initially be disabled and *lp* will be rejecting requests routed to it. If it is desired, allow *lp* to accept requests for xyz:

   ```
   /usr/lib/accept xyz
   ```

   This will allow requests to build up for xyz and they will be printed when it is enabled at a later time.

5. When it is desired for printing to occur, be sure that the printer is ready to receive output. For several printers, this means that the top of form has been adjusted and that the printer is on-line. As explained above in Section 9, enable printing to occur on xyz:

   ```
   enable xyz
   ```

   When requests have been routed to xyz, they will begin printing.

13.2 Login Terminals

Login terminals may also be used as LP printers. To do this for a Diablo 1640 terminal called abc, perform the following:

1. As explained above in Section 4.1, introduce printer abc to LP using the model 1640 interface program:

   ```
   $ /usr/lib/lpadmin −pabc −v/dev/null −m1640 −l
   ```

   Note that **/dev/null** is used as abc's device because we will specify the actual device each time that abc is enabled. This device may be different from day to day. When abc is created, it will initially be disabled and *lp* will be rejecting requests routed to it. If it is desired, allow *lp* to accept requests for abc:

   ```
   /usr/lib/accept abc
   ```

   This will allow requests to build up for abc and they will be printed when it is enabled at a later time. It is not advisable to enable abc for printing, however, until the following steps have been taken.

2. Log the terminal in if this has not already been done.

3. Assuming the *tty*(1) command reports that this terminal is /dev/tty02, associate this device with printer abc:

   ```
   $ /usr/lib/lpadmin -pabc -v/dev/tty02
   ```

   Note that *lpadmin* may be used only by an LPA. If it is desired for other users to routinely perform this step, then an LPA may establish a program owned by **lp** or by **root** with set-user-id permission that performs this function.

4. When it is desired for printing to occur, be sure that the printer is ready to receive output. For several printers, this means that the top of form has been adjusted. As explained above in Section 9, enable printing to occur on abc:

   ```
   enable abc
   ```

   When requests have been routed to abc, they will begin printing.

5. When all printing has stopped on abc or when you want it back as a regular login terminal, you may prevent it from printing more output:

   ```
   $ disable abc
   printer "abc" now disabled
   ```

   If abc is enabled when UNIX is rebooted or when *lpsched* is restarted, it will be disabled automatically.

14. SUMMARY

The administrative functions of the LP Administrator have been described in detail. They include configuring and re-configuring LP, maintaining printer interface programs, accepting, rejecting and moving print requests, stopping and starting the LP scheduler and enabling and disabling printers. LP offers administrators the following advantages over other centrally supported printer packages:

- Printers may be grouped into classes.
- LP may be configured to meet the needs of each site.
- Administrators may supply interface programs to format output in any way desirable.
- LP functions are performed by simple commands and not by hand.

REFERENCE

[1] Kliegman, J. R. *The Implementation of the LP Spooling System*, Bell Laboratories.

UNIX Operations Manual

A. G. Petruccelli

Bell Laboratories
Piscataway, New Jersey 08854

ABSTRACT

This manual contains a complete description of PDP-11/45 and-11/70 console operations, step-by-step instructions for normal operator functions, as well as descriptions of the UNIX† system console error messages. Console operating instructions for the VAX 11/780 can be found in *vaxops*(8) in the *UNIX User's Manual*.

The information in this manual was gathered from personal experience, the *UNIX User's Manual*, Digital Equipment Corporation (DEC) hardware manuals, and papers in *Documents for UNIX*.

Because this manual is intended to be as general as possible, it is suggested that each location add specific information about:

- Hardware configuration.
- Telephone line configuration.
- Specific logging and record-keeping practices.
- Contacts for hardware and software problems.
- Site-dependent diagnostic procedures.

† UNIX is a trademark of Bell Laboratories.

HARDWARE OPERATIONS—PDP-11/45, 11/70

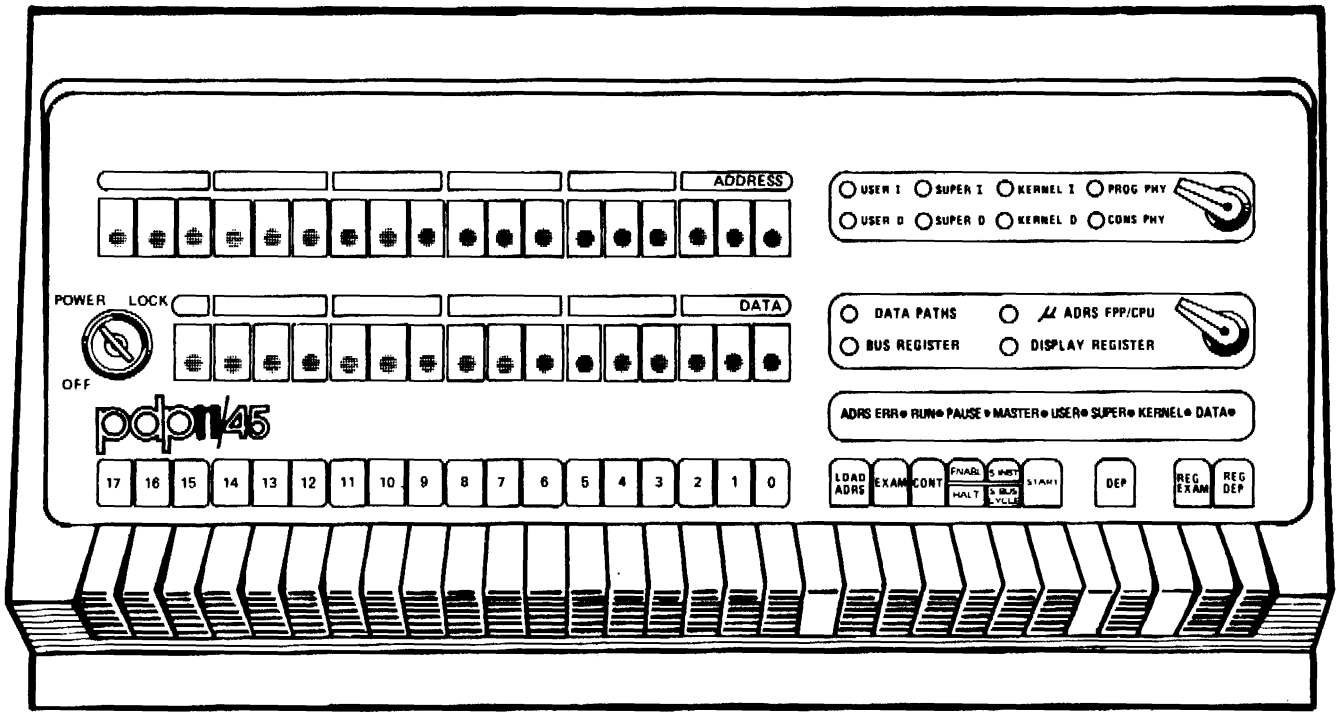


Figure 1. PDP-11/45 CONSOLE

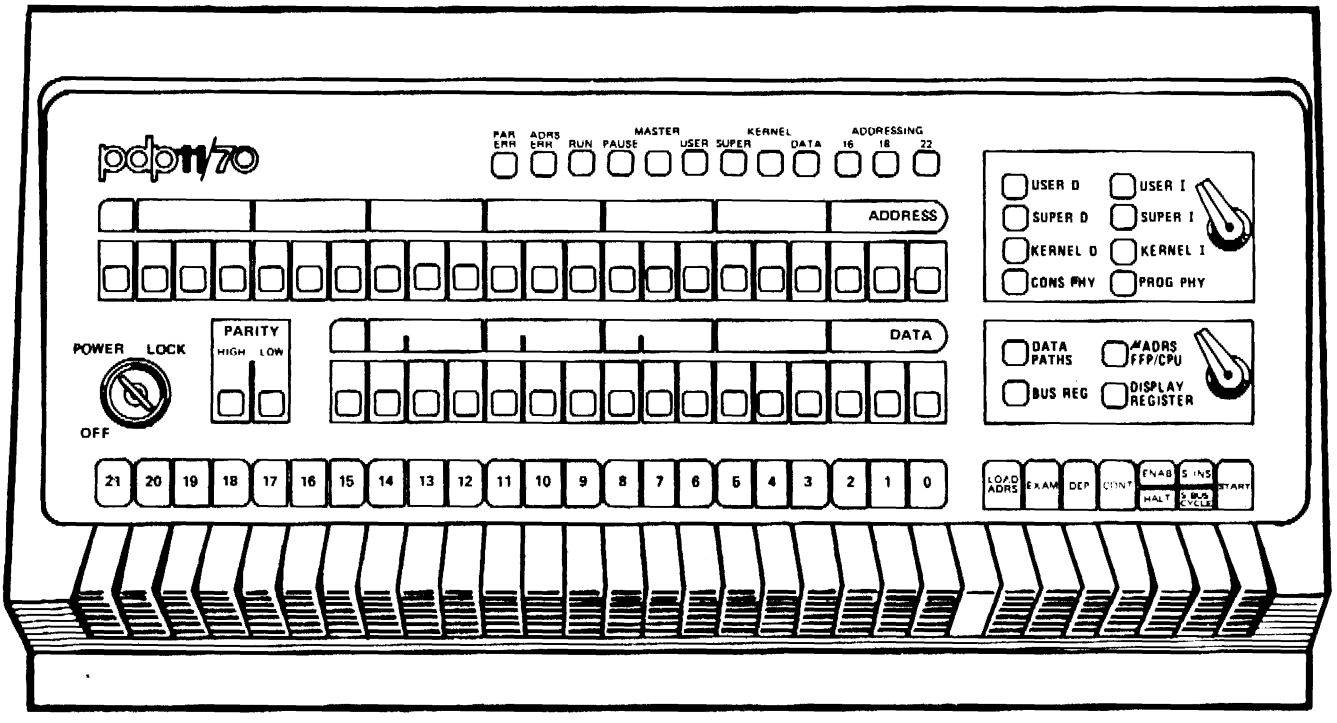


Figure 2. PDP-11/70 CONSOLE

INTRODUCTION

The following documentation is primarily intended to describe the PDP-11/70 console and its operation. Differences for the PDP-11/45 appear within brackets "[]". Cases that are applicable to only one of the two systems are clearly labeled as such.

CONSOLE DESCRIPTION

The console is composed of the following:

1. Power Key Switch (OFF/POWER/LOCK).
2. ADDRESS Register — 22-bit [18-bit] Display.
3. DATA Register — 16-bit Display.
4. PARITY bit HIGH byte & LOW byte Indicator Lights (11/70 only).
5. Switch Register — 22 [18] switches.
6. Error Lights.

   ADRS ERR (Address Error)
   PAR ERR (Parity Error, 11/70 only)

7. Processor State Lights (7 indicators).

   RUN
   PAUSE
   MASTER
   USER
   SUPER
   KERNEL
   DATA

8. Mapping Lights (11/70 only).

   16 BIT
   18 BIT
   22 BIT

9. ADDRESS Display Select Switch (8 positions).

   USER I (Virtual)
   USER D (Virtual)
   SUPER I (Virtual)
   SUPER D (Virtual)
   KERNEL I (Virtual)
   KERNEL D (Virtual)
   PROG PHY (Program Physical)
   CONS PHY (Console Physical)

10. DATA Display Select Switch (4 positions)

    DATA PATHS
    BUS REGISTER
    μ ADRS FPP/CPU (Micro-program Addresses)
    DISPLAY REGISTER

11. Lamp Test Switch.

12. Control Switches.

    LOAD ADRS (Load Address)
    EXAM (examine)
    DEP (deposit)
    CONT (continue)
    HALT/ENABLE [ENABL]
    S INST/S BUS CYCLE (single instruction/single bus cycle)
    START
    REG EXAM (Register Examine, 11/45 only)
    REG DEP (Register Deposit, 11/45 only)

CONSOLE OPERATION—LAMP TEST SWITCH

The Lamp Test Switch is an unlabeled, white switch located between the Switch Register and the **LOAD ADRS** Switch. When the Lamp Test Switch is raised, all console indicator lights should go on. An indicator which does not light is defective and should be replaced.

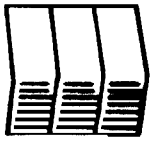
CONSOLE OPERATION—POWER KEY

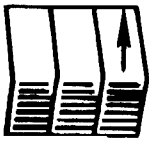
The Power Key controls power to the CPU[1] and has three positions:

OFF	Power to the processor is **OFF**.
POWER	Power to the processor is **ON**, and all console switches function normally. This is the *normal* position while UNIX is running.
LOCK	Power to the processor is **ON**, but the 7 control switches **LOAD ADRS** through **START** are disabled. All other switches are functional.

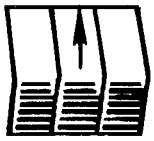
CONSOLE OPERATION—SWITCH REGISTER

The Switch Register consists of 22 [18] switches labeled 0 through 21 [17] from right to left (numbers correspond to bit positions). They are used to manually enter both addresses and data into the processor. To enter an address such as 165000_8, the switches must be divided into groups of three, starting from the right. Bits 0-2 in the first group, bits 3-5 in the second, 6-8 in the third, 9-11 in the fourth, 12-14 in the fifth, etc. Each group of 3 switches is used to indicate an octal digit; thus, a number can be represented on the switches as follows:

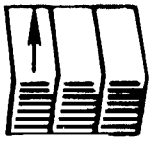
zero  All three switches down.

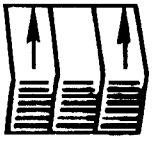
one  Right switch up.

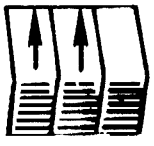
1. Central Processing Unit.

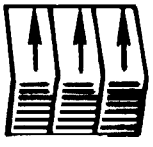
two  Middle switch up.

three  Middle and right switches up.

four  Left switch up.

five  Left and right switches up.

six  Left and middle switches up.

seven  All three switches up.

The arrows in Figures 3 and 4 depict which switches should be up to enter the address 165000_8 on the 11/70 and address 173020_8 on the 11/45 respectively.

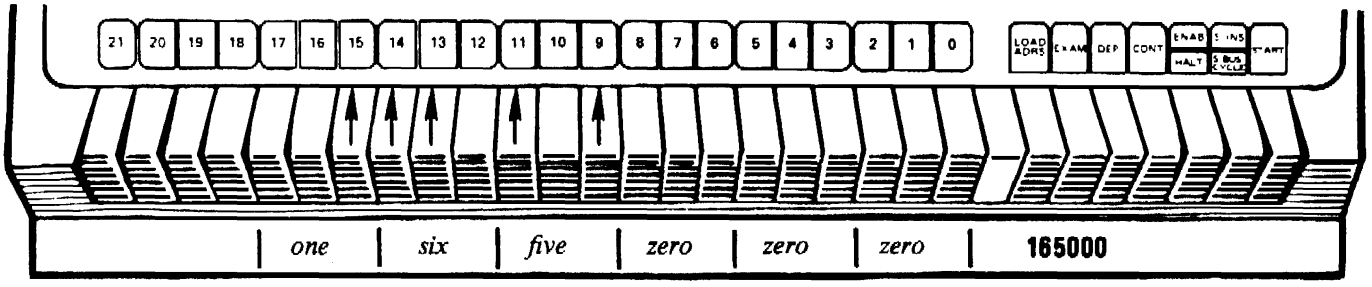


Figure 3. Address 165000_8 on the PDP-11/70.

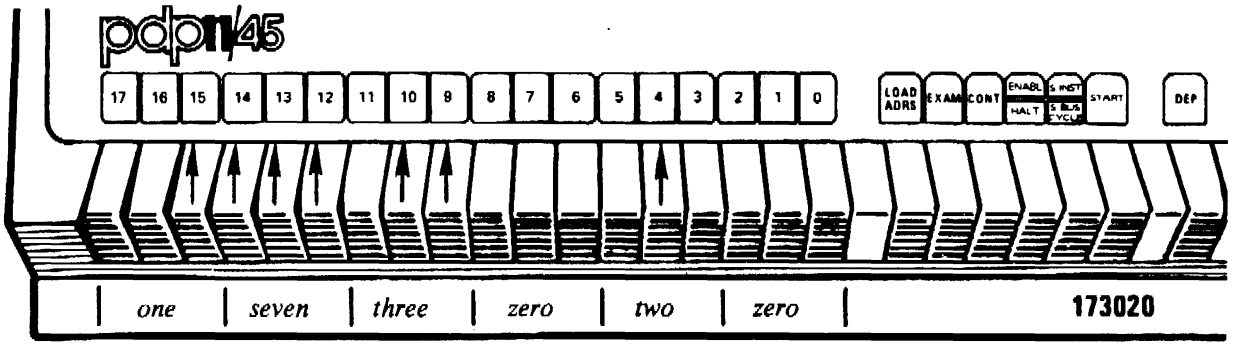


Figure 4. Address 173020_8 on the PDP-11/45.

CONSOLE OPERATION—CONTROL SWITCH FUNCTIONS

LOAD ADRS (Load Address)
When the **LOAD ADRS** Switch is depressed, the contents of the Switch Register are loaded into the Address Display. The address displayed in the Address Display Lights depends on the position of the Address Select Switch.

EXAM (Examine)
Depressing the **EXAM** Switch causes the contents of the current location, specified in the Address Display, to be displayed in the **DATA** Display Register when the Data Select Switch is in the **DATA PATHS** position.[2]

DEP (Deposit)
Raising the **DEP** Switch causes the current contents of the Switch Register to be deposited into the address specified by the current contents of the Address Display.

2. The address in the Address Display will be mapped or unmapped depending on the position of the Address Select Switch. The location shown in the Address Display Lights is also a function of that switch. See Section 11.4 of [1] for more information.

CONT (Continue)
Depressing the **CONT** Switch causes the CPU to resume execution. The **CONT** Switch has no effect when the CPU is in **RUN** state.

HALT/ENABLE [ENABL]
The **HALT/ENABLE [ENABL]** Switch is a two position switch used to stop machine execution or to enable the system to run.

S INST/S BUS CYCLE (Single Instruction/Single Bus Cycle)
This switch affects only the operation of the **CONT** Switch. It controls whether the machine stops after instructions or bus cycles.[3] The position of this switch has no effect unless the **HALT/ENABLE [ENABL]** Switch is in the **HALT** position. It is used chiefly for debugging. See Section 11.7 of [1] for more information.

START
The functions of the **START** Switch depend upon the setting of the **HALT/ENABLE [ENABL]** Switch. If the CPU is in the **HALT** position, the processor is reset. If in the **ENABLE [ENABL]** position, execution is started unless it is already in the **RUN** state.

REG EXAM (Register Examine, 11/45 only)
Depressing the **REG EXAM** Switch causes the contents of the General Purpose Register specified by the low order five bits of the Bus Address Register to be displayed in the Data Display Register. See Section 9.6.8 of [2] for interpretation of these contents.

REG DEP (Register Deposit, 11/45 only)
Raising the **REG DEP** Switch causes the contents of the Switch Register to be deposited into the General Purpose Register specified by the current contents of the CPU Bus Address Register. The CPU Bus Address Register should have been previously loaded by a **LOAD ADRS** operation according to the Switch Register settings described in **REG EXAM** above.

CONSOLE OPERATION—ADDRESS SELECT KNOB

The Address Select Knob has 8 positions for observing the address of data being examined or deposited. These positions reference virtual or physical memory as described below:

VIRTUAL	The six positions: **USER I, USER D, SUPER I, SUPER D, KERNEL I,** and **KERNEL D** indicate the current address as a 16-bit Virtual address when the Memory Management Unit is turned on (i.e. UNIX is running), otherwise it indicates the true 16-bit Physical Address.[4] These positions are generally used for debugging.
PROG PHY	This position displays the 22-bit [18-bit] Physical Address of the current bus cycle that was generated by the Memory Management Unit. This address is generally used for debugging.
CONS PHY	This position displays a 22-bit [16-bit] Physical Address to be used for console operations such as **LOAD ADRS, EXAM,** and **DEP**. This is the *normal* position while UNIX is running.

3. The *bus* (or UNIBUS) is the primary control and communications path connecting most of the PDP-11 system's components and peripherals.

4. These positions make it convenient to examine and change programs which are subject to relocation, without requiring any knowledge of where they have actually been relocated in physical memory. See Section 9.6.8, page 9-21 of [2] for more details. See Section 6.4 of [1] or Chapter 10 of [2] for more information on the Memory Management Unit.

CONSOLE OPERATION–DATA SELECT KNOB

The contents of the 16-bit Data Display Register are controlled by the following positions of the Data Select knob:

DATA PATHS	The *normal* display mode. This position enables examined or deposited data to be shown in the Data Display.
BUS REG	The internal CPU register used for bus cycles.
μADRS FPP/CPU	The ROM[5] address, FPP[6] control micro-program (bits 15 to 8) and the CPU control micro-program (bits 7 to 0).
DISPLAY REGISTER	The contents of the Display Register. This has an address of 17 777 570_8.

CONSOLE OPERATION–STATUS INDICATOR LIGHTS

Error Indicators

PAR ERR	(11/70 only) Lights to indicate a parity error during a reference to memory.
ADRS ERR	Lights to indicate any of the following addressing errors: • Reference to non-existent memory. • Access control violation. • Reference of unassigned memory pages.

Processor State

RUN	The CPU is executing program instructions. If the instruction being executed is a *wait* instruction, the **RUN** light will be on.
PAUSE	The CPU is inactive because the current instruction execution has been completed as far as possible without more data from the UNIBUS or memory or the CPU is waiting to regain control of the UNIBUS (UNIBUS mastership).
MASTER	The CPU is in control of the UNIBUS (UNIBUS Master only when it needs the UNIBUS).

Mode

USER	The CPU is executing program instructions in **USER** mode.
SUPER	The CPU is executing program instructions in **Supervisor** mode.
KERNEL	The CPU is executing program instructions in **KERNEL** mode.
DATA	If on, the last memory reference was to **D** (data) address space in the current CPU mode. If off, the last reference was to **I** (instruction) address space.

5. Read Only Memory.
6. Floating Point Processor.

Address (11/70 only)

16-bit	Lights when the CPU is using 16-bit mapping.
18-bit	Lights when the CPU is using 18-bit mapping.
22-bit	Lights when the CPU is using 22-bit mapping. This should be lit when running UNIX.

CONSOLE OPERATION—STARTING AND STOPPING

Starting

While the **HALT/ENABLE [ENABL]** Switch is in the **HALT** position (down), depress the **START** Switch to reset the processor. At this time, an address can be entered on the Switch Register as described above. After the correct switches have been lifted (check the **ADDRESS** Register Display Lights), depress the **LOAD ADRS** Switch to load that address as the starting point of execution, lift the **HALT/ENABLE [ENABL]** Switch to the **ENABLE [ENABL]** position, then depress the **START** Switch to commence execution. Once execution has begun, depressing the **START** Switch again has no effect.

Stopping

To halt execution of the processor, depress the **HALT/ENABLE [ENABL]** Switch to the **HALT** position. Processing will cease, but the contents of all memory locations will be retained. The switch can then be lifted to the **ENABLE [ENABL]** position with no effect on the system.

Continuing

After the computer has been stopped, execution can be resumed from the point at which it was halted by using the **CONT** Switch. The function of the **CONT** Switch depends on the position of the **HALT/ENABLE [ENABL]** Switch:

MODE	POSITION	USAGE
ENABLE [ENABL]	Up	CPU resumes normal execution.
HALT	Down	This mode is used for debugging purposes and forces execution of only a single instruction or a single bus cycle. See Section 11.7 of [1] for more details.

BOOT PROCEDURES

INTRODUCTION

The object of the boot procedure is to load a copy of the UNIX operating system, from tape or disk, into memory and execute it. This procedure can be easily facilitated via an optionally supplied Digital Equipment Corporation (DEC) hardware bootstrap loader. Depending upon which bootstrap loader is on your system, if any, the address of a dedicated routine can be loaded via the Switch Register and execution started. If your configuration does not include this device, the boot procedure must be manually entered via the Switch Register. See *romboot*(8) of [3] for program listings.

Throughout the remainder of this section, the symbol **<cr>** is used to denote a carriage return key at the terminal and the symbol *CSW* represents the Console Switches.

BOOTING FROM A ROM

The following procedure is used when booting from a ROM:

1. The Power Key Switch should be in the **POWER** position.
2. The Address Select Knob should be in the **CONS PHY** position.
3. The Data Select Knob should be in the **DATA PATHS** position.
4. Ensure the **HALT/ENABLE [ENABL]** Switch is in the **HALT** (down) position.
5. Depress the **START** Switch to reset the processor.
6. Set the *CSW* to the address of your ROM bootstrap loader procedure (i.e., 165000_8, 173020_8, etc.). If you don't know which address, ask your Customer Engineer (CE).
7. Depress the **LOAD ADRS** Switch to deposit this address into the Switch Register. Ensure the address was loaded correctly by checking the contents of the Address Display Register.
8. Depending upon the ROM, you may have to set the *CSW* to another address specifying from which device you wish to boot (i.e., 000070_8, 000060_8). Do **NOT** depress the **LOAD ADRS** Switch again, the bootstrap procedure will read this address.
9. Lift the **HALT** Switch to the **ENABLE [ENABL]** position.
10. Depress the **START** Switch to commence execution.
11. A "**#**" will be printed at the console terminal. You type a **0**, UNIX reponds with an =, and you type **unix** followed by a carriage return.

    #0=unix<cr>

    If UNIX was booted properly, four lines of information about the running system will be printed:

    - The current operating system.
    - The available user memory.
    - The system's name.
    - The environment mode (single-user).

MANUAL BOOT PROCEDURE

If your configuration does not include a hardware bootstrap loader, you will have to toggle the boot program into the processor via the *CSW*. The *romboot*(8) manual page in [3] contains program listings for booting off of a variety of devices. The procedure below for manually booting off of an RP04 disk drive will illustrate how to enter one of these programs:

1. Ensure the Power Key is in the **POWER** position.
2. The Address Select Knob must be in the **CONS PHY** position.
3. The Data Select Knob must be in the **DATA PATHS** position.
4. Ensure the **HALT/ENABLE [ENABL]** Switch is in the **HALT** (down) position.
5. Depress the **START** Switch to reset the processor.
6. Choose an arbitrary starting address to begin loading the program. This address must not be too low because the program's execution will overwrite it, and it cannot be too high because the processor will not be able to access it (in the memory management area of memory). The address 004000_8 (only switch 11 up) works well.
7. Set the *CSW* to this starting address and depress the **LOAD ADRS** Switch. You can now begin entering the program.

   Set *CSW* to 012700, lift **DEP** Switch.
   Set *CSW* to 176700, lift **DEP** Switch.
   Set *CSW* to 012720, lift **DEP** Switch.
   Set *CSW* to 000021, lift **DEP** Switch.
   Set *CSW* to 012760, lift **DEP** Switch.
   Set *CSW* to 010000, lift **DEP** Switch.
   Set *CSW* to 000030, lift **DEP** Switch.
   Set *CSW* to 010010, lift **DEP** Switch.
   Set *CSW* to 012740, lift **DEP** Switch.
   Set *CSW* to 000071, lift **DEP** Switch.
   Set *CSW* to 105710, lift **DEP** Switch.
   Set *CSW* to 002376, lift **DEP** Switch.
   Set *CSW* to 005007, lift **DEP** Switch.

   NOTE: The above octal digits represent the program for booting off of an RP04 disk drive only. For any other device, you must use the appropriate program listed in *romboot*(8) in [3].
8. You can check to be sure the program was entered correctly by setting the *CSW* to your starting address (e.g., 004000_8), depressing the **LOAD ADRS** Switch, and depressing the **EXAM** Switch. The first octal digit you entered (012700) should appear in the Data Display Register. By subsequent use of the **EXAM** Switch, the entire program can be listed for inspection.
9. After you are sure the program was entered correctly, reload the starting address (e.g., 004000_8) by setting the *CSW* and depressing the **LOAD ADRS** Switch.
10. Lift the **HALT** Switch to the **ENABLE [ENABL]** position.
11. Depress the **START** Switch.
12. A "**#**" will be printed at the console terminal. You type a **0**, UNIX reponds with an =, and you type **unix** followed by a carriage return.

    #0=unix<cr>

    If UNIX was booted properly, four lines of information about the running system will be printed:

    - The current operating system.
    - The available user memory.
    - The system's name.
    - The environment mode (single-user).

OPERATOR INSTRUCTIONS

INTRODUCTION

There are two main modes of operation of a UNIX system: Single-User and Multi-User.

When in Single-User mode, all dial-up ports and hard-wired terminals are disabled and only the console terminal may interact with the processor. This mode of operation enables any changes necessary to be made to the system without any other processing taking place.

Multi-User is the mode in which UNIX is normally run.

SINGLE USER ENVIRONMENT

After successfully booting the UNIX Operating System, as described in **BOOT PROCEDURES** within this document, a "**#**" will be typed as a prompt to indicate that the system is ready to receive commands. You may then type any of the commands available followed by a **<cr>**. When the system has completed execution of the command, it will prompt with the "**#**" again on the next line. The Single User environment is used primarily to do any system maintenance, modification, or repair operations to prepare the system for multi-user mode. The typical sequence of commands to bring the system up into multi-user mode are:

- fsck −t /tmp/junk
- date *MMddhhmmyy*
- init 2

Fsck

This program will interactively repair any damaged file systems that result from a crash of the operating system. It is also useful to ensure that the file systems have no damage before going into multi-user mode or taking file saves. Usually, you will want to respond "yes" to all the prompts; however, in the event of a system crash, the damage may be extensive enough to warrant recovery from a backup pack. The procedure for this is discussed in **FILE SAVES** in this document. The −t option is used to eliminate a prompt for the name of a scratch file if the file system is large. See *fsck* (1M) of [3] for details on the various options available and [4] for a description of all the different errors that can occur.

An example of a check of a consistent file system is illustrated below:

```
# fsck /dev/rrp61

/dev/rrp61
File System: usr Volume: p0603

** Phase 1 − Check Blocks and Sizes
** Phase 2 − Check Pathnames
** Phase 3 − Check Connectivity
** Phase 4 − Check Reference Counts
** Phase 5 − Check Free List
2441 files 16547 blocks 31889 free
#
```

A file system that has experienced some damage can be repaired interactively as shown below. The y is the operator response.

```
# fsck /dev/rrp60

/dev/rrp60
File System: fs1 Volume: p0603

** Phase 1 - Check Blocks and Sizes
POSSIBLE FILE SIZE ERROR I=2500

** Phase 2 - Check Pathnames
** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Counts
UNREF FILE I=2500  OWNER=255 MODE=100755
SIZE=0 MTIME=Dec 31 19:00 1969
CLEAR? y

** Phase 5 - Check Free List
2441 files 16547 blocks 889 free

***** FILE SYSTEM WAS MODIFIED *****
#
```

All mountable file systems should be listed in the file **/etc/checklist** which **fsck** uses, and these file systems checked each time the system is rebooted.

WARNING: Never execute **fsck** on an already mounted file system; it will have a bad effect since you are repairing only the physical disk. The only exception to this is the **root** file system which is always mounted.

An example of repairing the **root** file system follows:

```
# fsck /dev/rp0

/dev/rp0
File System: root Volume: p0001

** Phase 1 - Check Blocks and Sizes
POSSIBLE FILE SIZE ERROR I=416

POSSIBLE FILE SIZE ERROR I=610

POSSIBLE FILE SIZE ERROR I=614

POSSIBLE FILE SIZE ERROR I=618

POSSIBLE FILE SIZE ERROR I=625

** Phase 2 - Check Pathnames
** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Counts
UNREF FILE I=416  OWNER=uucp MODE=100400
SIZE=0 MTIME=Nov 20 16:23 1979
CLEAR? y

UNREF FILE I=610  OWNER=csw MODE=100400
SIZE=0 MTIME=Nov 20 16:26 1979
CLEAR? y
```

```
UNREF FILE I=625  OWNER=cath MODE=100400
SIZE=0 MTIME=Nov 20 16:26 1979
CLEAR? y

FREE INODE COUNT WRONG IN SUPERBLK
FIX? y

** Phase 5 - Check Free List
1 DUP BLKS IN FREE LIST
BAD FREE LIST
SALVAGE? y

** Phase 6 - Salvage Free List

585 files 5463 blocks 4223 free

***** BOOT UNIX (NO SYNC !) *****
```

At this time the processor should be halted and the system rebooted.

Date

Each time the system is rebooted, the software clock must be reset to the correct time of day. The date should only be set once and only in single-user mode. This will prevent confusion when the accounting routines run. The format for setting the date is:

date *MMddhhmmyy*

where:

MM	are the two digits of the month;
dd	are the two digits of the day;
hh	are the two digits of the hour on a 24-hour clock;
mm	are the two digits of the minute;
yy	are the (optional) last two digits of the year;

An example of how to set the date is:

date 0601073079

which would set the date for:

Fri Jun 1 07:30:00 EDT 1979

More information concerning the **date** command can be found on *date* (1) in [3].

Init 2

After you have performed the above consistency checks on the file systems and set the date, the mode of the operating system can be changed to multi-user. This is accomplished by executing the command: **/etc/init 2**. This command activates processes that: allow users to log on to the system, turn on the accounting and error logging, mount any indicated file systems, and start the **cron** and any indicated daemons. The operator may have to manually flip the toggles or pop the buttons on the data sets, depending on what type of data set your site has, to allow users to log in. You can now type a *Ctrl/d* character to log off the console terminal and log back in as a normal user.

MULTI-USER ENVIRONMENT

This mode results from the execution of the command: **/etc/init 2**. A user is permitted to access all mounted file systems and execute all available commands. In this mode, an operator can perform file restore procedures and take periodic status checks of the system. Some of these periodic status checks can include:

- A check of free blocks (**df**) remaining on all mounted file systems to ensure a file system does not run out of space.
- A check on rje (**rjestat**).
- A check on **mail** to root or whatever login receives requests for file restores.
- A check on the number of users on the system (**who**).
- A check of all running processes (**ps −eaf** or **whodo**) to determine if there is some process using an abnormally large amount of CPU time.

OPERATOR DUTIES

INTRODUCTION

This section is meant to serve as a guide to duties normally performed by computer operators. These duties do not represent what an operator's job duties are; they merely outline the general procedures necessary to ensure that users on the system remain contented.

FILE SAVES

Unless timely copies of the file systems are saved, a major system crash could devastate the system's user community.

Almost nothing is worse than working on a project and, just as you are about completed, having the system crash from a lightning storm somewhere; losing the file and all the work you've completed. What *is* worse is when you request a file restore and find out that the last file save was a week ago and that you have nothing to show for a solid week's work.

The easiest way to prevent this problem is to take daily file saves. Then, at most, only a day's work will be lost.

There are two main ways to perform file saves: by disk and by tape. Most sites utilize **volcopy** to perform these file save functions. See *volcopy*(1M) of [3] for more information on the options available and the use of this command. These procedures should normally be performed while in single-user mode, with the file system unmounted, to preclude any file system activity and subsequent damage on the saved copy.

Disk Save Procedures

Normally this is an automated procedure and is included as part of the site's local operating instructions. You must have at least two (2) disk drives, one of them a spare. The file system to be copied should be unmounted, except for the **root** file system, and an **fsck** executed to ensure consistency. For ease of mapping, file systems are normally saved in the same sections on the backup pack as they exist on the working pack. This is imperative if you are going to boot from the backup version. It is required that the **root** file system reside on section **0** of the pack. The file save procedure is illustrated below. For this example a save of the **root** file system will be made on the spare drive **3.** Operator response is indicated in bold type.

```
# volcopy root /dev/rrp0 p0001 /dev/rrp30 p0105
arg.(p0105) doesn't agree with to vol.()
Type 'y' to override:    y
warning! from fs(root) differs from to fs()
Type 'y' to override:    y
From: /dev/rrp0, to: /dev/rrp30? (DEL if wrong)
END: 6000 blocks.
#
```

You should conclude this procedure by executing **fsck** on the saved copy, just to be sure. Again, a backup of a file system that is corrupted is almost as bad as no save at all.

Tape Save Procedures

Tape saves are necessary for long term storage or for regular saves if you do not have a spare disk drive. Tapes must be labeled before a file save with **volcopy** can be accomplished. If the save will require two or more tapes, *both* tapes must be labeled *before* the **volcopy** is started. To determine the number of tapes the file save will require, try:

```
# volcopy -bpi1600 -feet2400 filesys /dev/rrp?? volume /dev/rmt1 t0001
You will need 1 reels.
From: /dev/rrp??, to: /dev/rmt1? (DEL if wrong)  Hit DEL
#
```

The above procedure assumes you are using 2400 feet reels, and **/dev/rmt1** indicates 1600 bytes-per-inch density. To accomplish the labeling, follow the example below for **/usr**. It is assumed that **t0001** is the tape volume label. If two or more tapes are required, they should be labeled consecutively both externally and internally. The external label should indicate which sequence number the tape is of the set for the file system. Note the use of the **−n** option. Unless you use this option on an unlabeled tape, the program will scan the entire reel looking for a label to change before it rewinds and labels the beginning. This can be very time consuming on 2400 feet reels.

```
# labelit /dev/rmt1 usr t0001 −n
Skipping label check!
NEW fsname = usr, NEW volume = t0001 −− DEL if wrong!!
#
```

After the tapes are labeled, you should then check the disk file system for errors with **fsck**. The actual copy is accomplished much the same as from disk to disk. The only difference is you may have to respond to more questions if the options of **−bpi** and **−feet** are not included on the command line of **volcopy**.

```
# volcopy usr /dev/rrp1 p0001 /dev/rmt1 t0001
Enter size of reel in feet for <t0001>:   2400
Tape density? (i.e., 800 | 1600 | 6250)?   1600
You will need 1 reels.
From: /dev/rrp1, to: /dev/rmt1? (DEL if wrong)
END: 35000 blocks.
#
```

FILE RESTORES

If your installation includes daily file saves as a normal routine, these backup versions of the file systems can provide a user good insurance against the loss of a lot of previous work due to a system crash and subsequent file system damage.

Restoring from Disk

When a request is made to restore a file from a backup pack, the operator should locate that pack and determine on which section the requested file system resides. Place that pack on a spare drive and power on the drive. You may choose to mount the file system write protected by specifying the **−r** option of **mount**. At the console terminal the operator should log onto the system as **root**. The following example shows the procedure for restoring the file **/usr/adm/acct/sum/tacct** from a previous backup pack. For this example, drive **4** is a spare drive and **/usr** is on section **1** of the backup pack.

```
# mount  /dev/rp41  /bck −r
WARNING!! − mounting: <usr> as </bck>
# ls −l /bck/adm/acct/sum/tacct   (To verify file existence and identify owner.)
−rw−rw−r−− 1 adm      3216 Oct 3 03:29 /usr/adm/acct/sum/tacct
# cp  /bck/adm/acct/sum/tacct  /usr/adm/acct/sum/tacct
# chown adm /usr/adm/acct/sum/tacct
# umount /dev/rp41
#
```

It is usually a good practice for the operator performing the file restore to **mail** a message to the requester upon its completion. The procedure for this is:

```
# mail user
I have restored the file /usr/adm/acct/sum/tacct
from Friday's backup.
operator's initials
.
#
```

Restoring from Tape

If the file does not exist on any of the backup packs or if your installation does not perform disk file saves, then you will have to recover the file from a tape save. It is assumed that tape saves have been performed in the same manner as disk saves, i.e., with **volcopy**. The subject of file saves is discussed in the section **FILE SAVES** within this document. In order to restore a file from tape, the whole file system must first be placed back on a spare section of the disk. The backup version can then be accessed in the same way as described in **Restoring from Disk** within this document. For this example, it is assumed that the **usr** file system is the second file on the tape and that section **5** of disk drive **0** is a spare section on that disk. It is also assumed that the tape drive has 1600bpi capability; if not, a similar procedure can be followed for 800bpi recorded tapes.

```
(mount tape on tape drive 0)
# echo < /dev/mt4   (space past first file on tape, no rewind)
# volcopy usr /dev/rmt1 t0001 /dev/rrp5 p0001
Enter size of reel in feet for <t0001>:   2400
Tape density? (i.e., 800 | 1600 | 6250)?   1600
You will need 1 reels.
From: /dev/rmt1, to: /dev/rrp5? (DEL if wrong)
END: 35000 blocks.
# mount /dev/rp5 /bck
WARNING!! - mounting: <usr> as </bck>
# cp /bck/adm/acct/sum/tacct /usr/adm/acct/sum/tacct
# umount /dev/rp5
#
```

MESSAGE OF THE DAY

When a user logs into the system, part of the login procedure prints out a message of the day. This message can contain several lines of useful information to the user concerning scheduled down-time for hardware preventive maintenance (PM), clean up messages for space-low file systems, or any other useful warnings to which users may need to be alerted. The trick to maintaining this file is to keep it short and to the point. A user does not want to wait ten minutes while eloquent and wordy dialogue is spewed from the terminal before he or she can begin working.

The contents of this message is stored in the file **/etc/motd**. You may change the contents of this file by using the UNIX text editor (see *ed*(1) in [3]). A sample of adding and deleting a line from this file is shown below.

```
# ed /etc/motd
26
p
9/23: Reboot at 5pm today.
d
a
9/24: Down for PM 1700-2100 on 9/30.
.
w
37
q
#
```

You can also remove the contents of the entire file by:

```
# cp /dev/null /etc/motd
#
```

SYSTEM SHUTDOWN

Whenever the system must be shutdown, such as for file saves or a reboot, the program **/etc/shutdown** should be used. This program is the graceful way to bring the system into single-user mode. You can specify the amount of grace period between sending a warning message out and actually shutting down. This grace period is the number of seconds of delay. You may, optionally, send your own message. A default message is sent to all logged in users if you don't type your own. The following shows an example of shutting the system down:

```
# /etc/shutdown 300              (5 minute grace period)
shutdown: you must be in the root directory (/) to use shutdown
# cd /
# /etc/shutdown 300

SHUTDOWN PROGRAM

Thu Sep  1 18:51:58 EST 1979

Do you want to send your own message? (y or n):  y
Type your message followed by ctrl d....
System coming down for filesaves!
Please log off.
(Cntl/d)

System coming down for filesaves!
Please log off.
(waits for 5 minutes)
SYSTEM BEING BROUGHT DOWN NOW ! ! !

Busy out (push down) the appropriate
phone lines for this system.

Do you want to continue? (y or n):  y
Error logging stopped
Hasp stopped
Process accounting stopped.

All currently running processes will now be killed.
```

```
Changing init states, continue (y or n): y
pwba
single-user

  PID  TTY  TIME  CMD
    0   ?  187:48 swapper
    1   ?    0:03 INIT 1
11061  co    0:04 -sh
25023  co    0:03 /etc/shutdown 300
25052  co    0:23 ps -eaf

Will a file save be done at this time?
Type either ( y or n ) : y
Want to run fsck at this time?
Type either ( y or n ) : y
fsck will now be executed on files in checklist
        :
        :
Halt the system when ready.
```

At the completion of this program you can either halt the system, start the file save routine, reboot the system, or bring it back to multi-user mode.

SYSTEM CRASH RECOVERY

An operating system is considered to have "crashed" when it halts itself without being asked to. The reason for the halt is often unknown and can be hardware failure or software related. It is important, for obvious reasons, to determine the nature of the crash so that it will not happen again. One way to do this is to take a dump of memory on tape so that debugging programs can later decipher what processing was going on at the time the crash occurred. The method for this is:

1. Mount a tape on drive 0 with a write ring in.
2. Set *CSW* to the address 000044_8 (switches 5 and 2 up).
3. Lift the **HALT** Switch to the **ENABLE [ENABL]** position.
4. Depress the **START** Switch.

When the tape has rewound, unmount it and affix a label with the date and time of the crash written on it. You should now attempt to reboot UNIX as described in **BOOT PROCEDURES** in this document. If the system fails to reboot, the operating system was probably damaged in the crash. Now is the time to pull that vital backup version of the **root** file system off the shelf and use it for the reboot.

When you have finally rebooted the system, it is likely to have a lot of file system damage. If this damage is extensive, you may have to restore the entire file system. An example is:

```
(mount the backup pack on a spare drive)
# volcopy fs1 /dev/rrp43 p0625 /dev/rrp3 p0601
From: /dev/rrp43, to: /dev/rrp3? (DEL if wrong)
END: 65000 blocks.
#
```

Be sure to run **fsck** on all the mountable file systems before setting the date and going to multi-user mode.

SYSTEM ERROR MESSAGES

INTRODUCTION

Sometimes before UNIX crashes, it has time to print some error messages and warnings. You may notice if you are logged in as a normal user, that the system will stop everything for a period of time while it is printing messages to the console terminal (remember to leave at least one console switch up!). There are a wide variety of messages that can occur but there are only two distinct types:

Fatal – System failure is imminent, and
Warning – Something is happening that may lead to a system failure.

SYSTEM ERROR MESSAGES–FATAL

panic: no clock
: Neither the KW11-L nor the KW11-P was found at their standard UNIBUS addresses.

panic: buffers
: Insufficient memory space was found when the system was attempting to allocate the non-addressable buffer pool.

panic: iinit
: An error occurred while the system was reading in the superblock of the root file system.

< loop at User location 6 >
: The initialization and line monitor program, /etc/init, cannot be executed.

panic: IO err in swap
: An unrecoverable error has occurred during a system swap operation.

panic: Out of swap
: Insufficient space was found on the system swap device when attempting to allocate buffering for the arguments to a process overlay.

panic: out of swap space
: Insufficient space was found on the swap device when attempting to swap out a process or a copy of a pure text image.

panic: trap
: An unexpected system fault has occurred. This message is preceded by the type of trap and the location of the currently running process.

death
: A system stack overflow has occurred, typically caused by repetitive interrupting of a faulty I/O device.

panic: parity
: A memory system error has occurred in the realm of the operating system address space. When this occurs in a User process, that process is terminated without a panic.

Timeout table overflow
: The system timeout table–used to implement software interrupts–has overflowed while attempting to add another entry.

panic: devtab
: The list header for the chain of buffers attached to a block type device cannot be found.

panic: blkdev
: The major device number of a block type device exceeds the number of such devices in the system. This error should have been detected earlier.

panic: no fs
: The superblock of a mounted file system cannot be found.

panic: no imt
: A mount point was not found in the system mount table when traversing a file system boundary.

panic: no procs
: A process table entry cannot be found during a process fork when it is known that an entry is available.

SYSTEM ERROR MESSAGES—WARNINGS

no space on dev major/minor
: The corresponding file system has run out of available free blocks.

Out of inodes on dev major/minor
: The corresponding file system contains no more free file control blocks.

bad block on dev major/minor
: A block number not in the valid range of available free blocks on a file system has been detected.

Bad free count on dev major/minor
: A corrupted free-list block has been detected while attempting to allocate a new block for a file.

bad count on dev major/minor
: The super block parameters for free blocks and inodes have become corrupted for this file system.

Inode table overflow
: The system file control block table has overflowed. An access to a currently unused file has failed.

No file
: The system file access control table has overflowed. A new reference to a file has failed.

Out of text
: The system shared text program control table has overflowed. An attempt to execute a currently unused shared text program has failed.

Power fail #
: A power fail condition has been detected. If power fail recovery has been specified in the system configuration, the initialization process will be informed.

Stopped
: Printed after a power fail condition if recovery has not been specified. The system is halted.

iaddress > 2^24
: When updating the file control block for a file, a block number in the inode was found to be greater than that permissible.

proc on q
: When making a process runnable, after the occurrence of a wakeup event, it was found that the process was already on the system run queue.

parity
: A system memory error has occurred. This message is followed by the contents of the low error address register, the high error address register, the memory system error register and the memory control register.

Stray interrupt at vector
: A device has interrupted through a vector not specified in the configuration description.

RP04/5/6 drive # not available
: The designated drive is no longer available for I/O operations due to an error condition.

hard err on RP04/5/6 rpds rperl rper2 rper3
: After a certain number of unsuccessful retry attempts, a hard error still exists on a drive access. The register contents are those of the drive status register and the 3 drive error registers.

DMC# lost block
: The buffer header for a dmc transfer operation claimed to be completed cannot be found.

RS03/4 not available
: An RS03 or RS04 drive is no longer accessible.

dzk: xint
: In a system with DZ11 multiplexors with KMC11 assist, a transmitter interrupt has occurred.

Hardware Error.
: When a hardware error occurs on a block type device, certain device dependent registers are displayed. The message is of the form:

    err on dev major/minor

  followed by:

    bn=# er=#, #

  This is the block number in error, followed by an error register and a control register whose contents can be interpreted in [5] under the appropriate device as shown below:

DEVICE	*ERROR REGISTER*	*CONTROL REGISTER*
RP06	RPER1	RPCS2
RS04	RSCS2	0
TE16	MTER	MTCS2
RP03	RPCS	RPDS
RK05	RKER	RKDS
RF11	RFCS	RFDAE

REFERENCES

[1] *PDP-11/70 Processor Handbook*, Digital Equipment Corporation, 1977-78.

[2] *PDP-11 04/34/45/55 Processor Handbook*, Digital Equipment Corporation, 1976-77.

[3] T. A. Dolotta, S. B. Olsson, and A. G. Petruccelli (eds.). *UNIX User's Manual*—Release 3.0, Bell Laboratories (June 1980).

[4] T. J. Kowalski. *FSCK—The UNIX File System Check Program*, Bell Laboratories (1979).

[5] *PDP-11 Peripherals Handbook*, Digital Equipment Corporation, 1978-79.

January 1981

CONTENTS

LIST OF FIGURES

FSCK—The UNIX File System Check Program

T. J. Kowalski

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The UNIX† 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. *Fsck* is frequently 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.

1. INTRODUCTION

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. No changes are made to any file system by *fsck* without prior operator approval.

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 heuristically sound corrective actions used by *fsck* (the Coast Guard to the rescue) is presented.

2. UPDATE OF THE FILE SYSTEM

Every working day hundreds of files are created, modified, and removed. Every time a file is modified, the UNIX operating system performs a series of file system updates. These updates, when written on disk, yield a consistent file system. To understand what happens in the event of a permanent interruption in this sequence, it is important to understand the order in which the update requests were probably being honored. Knowing which pieces of information were probably written to the file system first, heuristic procedures can be developed to repair a corrupted file system.

There are five types of file system updates. These involve the super-block, inodes, indirect blocks, data blocks (directories and files), and free-list blocks.

2.1 Super-Block

The super-block contains information about the size of the file system, the size of the inode list, part of the free-block list, the count of free blocks, the count of free inodes, and part of the free-inode list.

---

† UNIX is a trademark of Bell Laboratories.

The super-block of a mounted file system (the root file system is always mounted) is written to the file system whenever the file system is unmounted or a *sync* command is issued.

2.2 Inodes

An inode contains information about the type of inode (directory, data, or special), the number of directory entries linked to the inode, the list of blocks claimed by the inode, and the size of the inode.

An inode is written to the file system upon closure[1] of the file associated with the inode.

2.3 Indirect Blocks

There are three types of indirect blocks: single-indirect, double-indirect and triple-indirect. A single-indirect block contains a list of some of the block numbers claimed by an inode. Each one of the 128 entries in an indirect block is a data-block number. A double-indirect block contains a list of single-indirect block numbers. A triple-indirect block contains a list of double-indirect block numbers.

Indirect blocks are written to the file system whenever they have been modified and released[2] by the operating system.

2.4 Data Blocks

A data block may contain file information or directory entries. Each directory entry consists of a file name and an inode number.

Data blocks are written to the file system whenever they have been modified and released by the operating system.

2.5 First Free-List Block

The super-block contains the first free-list block. The free-list blocks are a list of all blocks that are not allocated to the super-block, inodes, indirect blocks, or data blocks. Each free-list block contains a count of the number of entries in this free-list block, a pointer to the next free-list block, and a partial list of free blocks in the file system.

Free-list blocks are written to the file system whenever they have been modified and released by the operating system.

3. CORRUPTION OF THE FILE SYSTEM

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

3.1 Improper System Shutdown and Startup

File systems may become corrupted when proper shutdown procedures are not observed, e.g., forgetting to *sync* the system prior to halting the CPU, physically write-protecting a mounted file system, or taking a mounted file system off-line.

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.

---

1. All in core blocks are also written to the file system upon issue of a *sync* system call.
2. More precisely, they are queued for eventual writing. Physical I/O is deferred until the buffer is needed by UNIX or a *sync* command is issued.

3.2 Hardware Failure

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.

4. DETECTION AND CORRECTION OF CORRUPTION

A quiescent[3] 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. A quiescent state is important during the checking of a file system because of the multi-pass nature of the *fsck* program.

When an inconsistency is discovered *fsck* reports the inconsistency for the operator to chose a corrective action.

Discussed in this section are how to discover inconsistencies and possible corrective actions for the super-block, the inodes, the indirect blocks, the data blocks containing directory entries, and the free-list blocks. These corrective actions can be performed interactively by the *fsck* command under control of the operator.

4.1 Super-Block

One of the most common corrupted items is the super-block. The super-block is prone to corruption because every change to the file system's blocks or inodes modifies the super-block.

The super-block and its associated parts are most often corrupted when the computer is halted and the last command involving output to the file system was not a *sync* command.

The super-block can be checked for inconsistencies involving file-system size, inode-list size, free-block list, free-block count, and the free-inode count.

4.1.1 File-System Size and Inode-List Size. 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 number of inodes must be less than 65,535. The file-system size and inode-list size are critical pieces of information to the *fsck* program. While there is no way to actually check these sizes, *fsck* can check for them being within reasonable bounds. All other checks of the file system depend on the correctness of these sizes.

4.1.2 Free-Block List. The free-block list starts in the super-block and continues through the free-list blocks of the file system. Each free-list block can be checked for a list count out of range, for block numbers out of range, and for blocks already allocated within the file system. A check is made to see that all the blocks in the file system were found.

The first free-block list is in the super-block. *Fsck* checks the list count for a value of less than zero or greater than fifty. It also checks each block number for a value of less than the first data block in the file system or greater than the last block in the file system. Then it compares each block number to a list of already allocated blocks. If the free-list block pointer is non-zero, the next free-list block is read in and the process is repeated.

When all the blocks have been accounted for, a check is made to see if the number of blocks used by the free-block list 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 free-block list, then *fsck* may rebuild it, excluding all blocks in the list of allocated blocks.

---

3. That is, unmounted and not being written on.

4.1.3 Free-Block Count. The super-block contains a count of the total number of free blocks within the file system. *Fsck* compares this count to the number of blocks it found free within the file system. If they don't agree, then *fsck* may replace the count in the super-block by the actual free-block count.

4.1.4 Free-Inode Count. The super-block contains a count of the total number of free inodes within the file system. *Fsck* compares this count to the number of inodes it found free within the file system. If they don't agree, then *fsck* may replace the count in the super-block by the actual free-inode count.

4.2 Inodes

An individual inode is not as likely to be corrupted as the super-block. However, because of the great number of active inodes, there is almost as likely a chance for corruption in the inode list as in the super-block.

The list of inodes is checked sequentially starting with inode 1 (there is no inode 0) and going to the last inode in the file system. Each inode can be checked for inconsistencies involving format and type, link count, duplicate blocks, bad blocks, and inode size.

4.2.1 Format and Type. Each inode contains a mode word. This mode word describes the type and state of the inode. Inodes may be one of four types: regular inode, directory inode, special block inode, and special character inode. If an inode is not one of these types, then the inode has an illegal type. Inodes may be found in one of three states: unallocated, allocated, and neither unallocated nor allocated. This last state indicates an incorrectly formatted inode. An inode can get in this state if bad data is written into the inode list through, for example, a hardware failure. The only possible corrective action is for *fsck* is to clear the inode.

4.2.2 Link Count. Contained in each inode is a count of the total number of directory entries linked to the inode.

Fsck verifies the link count of each inode by traversing down the total directory structure, starting from the root directory, calculating an actual link count for each inode.

If the stored link count is non-zero and the actual link count is zero, it means that no directory entry appears for the inode. 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 the stored link count is non-zero and the actual link count is zero, *fsck* may link the disconnected file to the *lost+found* directory. If the stored and actual link counts are non-zero and unequal, *fsck* may replace the stored link count by the actual link count.

4.2.3 Duplicate Blocks. Contained in each inode is a list or pointers to lists (indirect blocks) of all the blocks claimed by the inode.

Fsck compares each block number claimed by an inode to a list of already allocated blocks. If a block number is already claimed by another inode, 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 make a partial second pass of the inode list to find the inode of the duplicated block, because without examining the files associated with these inodes for correct content, there is not enough information available to decide which inode is corrupted and should be cleared. Most times, the inode with the earliest modify time is incorrect, and should be cleared.

This condition can occur by using a file system with blocks claimed by both the free-block list and by other parts of the file system.

If there is a large number of duplicate blocks in an inode, this may be due to an indirect block not being written to the file system.

Fsck will prompt the operator to clear both inodes.

4.2.4 Bad Blocks. Contained in each inode is a list or pointer to lists of all the blocks claimed by the inode.

Fsck checks each block number claimed by an inode for a value lower than that of the first data block, or greater than the last block in the file system. If the block number is outside this range, the block number is a bad block number.

If there is a large number of bad blocks in an inode, this may be due to an indirect block not being written to the file system.

Fsck will prompt the operator to clear both inodes.

4.2.5 Size Checks. Each inode contains a thirty-two bit (four-byte) size field. This size indicates the number of characters in the file associated with the inode. This size can be checked for inconsistencies, e.g., directory sizes that are not a multiple of sixteen characters, or the number of blocks actually used not matching that indicated by the inode size.

A directory inode within the UNIX file system has the directory bit on in the inode mode word. The directory size must be a multiple of sixteen because a directory entry contains sixteen bytes (two bytes for the inode number and fourteen bytes for the file or directory name).

Fsck will warn of such directory misalignment. This is only a warning because not enough information can be gathered to correct the misalignment.

A rough check of the consistency of the size field of an inode can be performed by computing from the size field the number of blocks that should be associated with the inode and comparing it to the actual number of blocks claimed by the inode.

Fsck calculates the number of blocks that there should be in an inode by dividing the number of characters in a inode by the number of characters per block (512) and rounding up. *Fsck* adds one block for each indirect block associated with the inode. If the actual number of blocks does not match the computed number of blocks, *fsck* will warn of a possible file-size error. This is only a warning because UNIX does not fill in blocks in files created in random order.

4.3 Indirect Blocks

Indirect blocks are owned by an inode. Therefore, inconsistencies in indirect blocks directly affect the inode that owns it.

Inconsistencies that can be checked are blocks already claimed by another inode and block numbers outside the range of the file system.

For a discussion of detection and correction of the inconsistencies associated with indirect blocks, apply iteratively Sections 4.2.3 and 4.2.4 to each level of indirect blocks.

4.4 Data Blocks

The two types of data blocks are plain data blocks and directory data blocks. Plain data blocks contain the information stored in a file. Directory data blocks contain directory entries. *Fsck* does not attempt to check the validity of the contents of a plain data block.

Each directory data block can be checked for inconsistencies involving directory inode numbers pointing to unallocated inodes, directory inode numbers greater than the number of inodes in the file system, incorrect directory inode numbers for "." and "..", and directories which are disconnected from the file system. In addition, the validity of the contents of a directory's data block is checked.

If a directory entry inode number points to an unallocated inode, then *fsck* may remove that directory entry. This condition probably occurred because the data blocks containing the directory entries were modified and written out, while the inode was not yet written out.

If a directory entry inode number is pointing beyond the end of the inode list, *fsck* may remove that directory entry. This condition occurs if bad data is written into a directory data block.

The directory inode number entry for "." should be the first entry in the directory data block. Its value should be equal to the inode number for the directory data block.

The directory inode number entry for ".." should be the second entry in the directory data block. Its value should be equal to 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* may replace them by the correct values.

Fsck checks the general connectivity of the file system. If directories are found not to be linked into the file system, *fsck* will link the directory back into the file system in the *lost+found* directory. This condition can be caused by inodes being written to the file system with the corresponding directory data blocks not being written to the file system.

4.5 Free-List Blocks

Free-list blocks are owned by the super-block. Therefore, inconsistencies in free-list blocks directly affect the super-block.

Inconsistencies that can be checked are a list count outside of range, block numbers outside of range, and blocks already associated with the file system.

For a discussion of detection and correction of the inconsistencies associated with free-list blocks see Section 4.1.2.

ACKNOWLEDGEMENT

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.

REFERENCES

[1] 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.

[2] Dolotta, T. A., Olsson, S. B., and Petruccelli, A. G., eds. *UNIX User's Manual*—Release 3.0 (June 1980).

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

APPENDIX: FSCK ERROR CONDITIONS

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 free-block list (possibly rebuilding it), and performs some cleanup.

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. 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 under initialization.

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.

C option?

C is not a legal option to *fsck*; legal options are **−y**, **−n**, **−s**, **−S**, **−t**, **−r**, **−q**, and **−D**. *Fsck* terminates on this error condition. See the *fsck*(1M) manual entry for further details.

Bad −t option

The **−t** option is not followed by a file name. *Fsck* terminates on this error condition. See the *fsck*(1M) manual entry for further details.

Invalid −s argument, defaults assumed

The **−s** option is not suffixed by 3, 4, or blocks-per-cylinder:blocks-to-skip. *Fsck* assumes a default value of 400 blocks-per-cylinder and 9 blocks-to-skip. See the *fsck*(1M) manual entry for more details.

Incompatible options: −n and −s

It is not possible to salvage the free-block list without modifying the file system. *Fsck* terminates on this error condition. See the *fsck*(1M) manual entry for further details.

Can't fstat standard input

Fsck's attempt to *fstat* standard input failed. This should never happen. *Fsck* terminates on this error condition. See a guru.

Can't get memory

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 default file system checklist file **F** (usually */etc/checklist*) 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

Fsck's request for statistics about the file system **F** failed. It ignores this file system and continues checking the next file system given. Check access modes of **F**.

F is not a block or character device

You have given *fsck* a regular file name by mistake. It ignores this file system and continues checking the next file system given. Check file type of **F**.

Can't open F

The file system **F** can not be opened for reading. It ignores this file system and continues checking the next file system given. Check access modes of **F**.

Size check: fsize X isize Y

More blocks are used for the inode list **Y** than there are blocks in the file system **X**, or there are more than 65,535 inodes in the file system. It ignores this file system and continues checking the next file system given. See Section 4.1.1.

Can't create F

Fsck's request to create a scratch file **F** failed. It ignores this file system and continues checking the next file system given. Check access modes of **F**.

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.

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.

UNKNOWN FILE TYPE I=I (CLEAR)

The mode word of the inode **I** indicates that the inode is not a special character inode, special character inode, regular inode, or directory inode. See Section 4.2.1.

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. See Section 4.2.4.

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. See Section 4.2.4.

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. See Section 4.2.3.

EXCESSIVE DUP BLKS I=I (CONTINUE)

There is more than a tolerable number (usually 10) of blocks claimed by other inodes. See Section 4.2.3.

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.

POSSIBLE FILE SIZE ERROR I=I

The inode I size does not match the actual number of blocks used by the inode. This is only a warning. See Section 4.2.5. If the **−q** option is used, this message is not printed.

DIRECTORY MISALIGNED I=I

The size of a directory inode is not a multiple of the size of a directory entry (usually 16). This is only a warning. See Section 4.2.5. If the **−q** option is used, this message is not printed.

PARTIALLY ALLOCATED INODE I=I (CLEAR)

Inode **I** is neither allocated nor unallocated. See Section 4.2.1.

Possible responses to the CLEAR prompt are:

YES de-allocate inode **I** by zeroing its contents.
NO ignore this error condition.

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** which 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. See Section 4.2.3.

5. PHASE 2: CHECK PATH-NAMES

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.

ROOT INODE UNALLOCATED. TERMINATING.

The root inode (usually inode number 2) has no allocate mode bits. This should never happen. The program will terminate. See Section 4.2.1.

ROOT INODE NOT DIRECTORY (FIX)

The root inode (usually inode number 2) is not directory inode type. See Section 4.2.1.

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. See Sections 4.2.3 and 4.2.4.

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. See Section 4.4.

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 NAME=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. See Section 4.4. If the directory entry is a non-empty directory, the REMOVE prompt will not appear, because *fsck* does not permit the removal of non-empty directories. The prompt will appear if the entry is not a directory and is non-empty. If the file system is not mounted and the **−n** option was not specified, the entry will be removed automatically if it is empty, regardless of whether or not it is a directory.

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. See Sections 4.2.3 and 4.2.4.

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. See Sections 4.2.3 and 4.2.4.

Possible responses to the REMOVE prompt are:

YES	the directory entry **F** is removed.
NO	ignore this error condition.

BAD BLK B IN DIR I=I OWNER=O MODE=M SIZE=S MTIME=T

A bad block was found in DIR inode **I**. This error message indicates that the user should, at a later time, either remove the directory inode if the entire block looks bad, or change (or remove) those directory entries that look bad. The block is checked to see whether it is a DUP block; if it is, *fsck* will print "IT'S A DUP BLOCK -- CLEAR MANUALLY".

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. See Sections 4.4 and 4.2.2. *Fsck* will force the reconnection of a non-empty directory *unless* a bad block was found on that directory in Phase 2.

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*(1M) manual entry for further details.

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*(1M) manual entry for further details.

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. See Sections 4.4 and 4.2.2.

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, or special files, unreferenced files and directories, bad and duplicate blocks in files and directories, and incorrect total free-inode counts.

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. See Section 4.2.2. If the **-n** option is not set and the file system is not mounted, empty files will not be reconnected and will be cleared automatically.

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.

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*.

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*.

(CLEAR)

The inode mentioned in the immediately previous error condition can not be reconnected. See Section 4.2.2.

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.

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. See Section 4.2.2.

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. See Section 4.2.2.

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. See Section 4.2.2.

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. See Sections 4.2.2 and 4.4. If the **−n** option is not set and the file system is not mounted, empty files will be cleared automatically.

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. See Sections 4.2.2 and 4.4. If the **−n** option is not set and the file system is not mounted, empty directories will be cleared automatically. Non-empty directories will not be 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. See Sections 4.2.3 and 4.2.4.

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. See Sections 4.2.3 and 4.2.4.

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. See Section 4.1.4. If the **-q** option is specified, the count will be fixed automatically in the super-block.

Possible responses to the FIX prompt are:

YES	replace the count in the super-block by the actual count.
NO	ignore this error condition.

8. PHASE 5: CHECK FREE LIST

This phase concerns itself with the free-block list. This section lists error conditions resulting from bad blocks in the free-block list, bad free-blocks count, duplicate blocks in the free-block list, unused blocks from the file system not in the free-block list, and the total free-block count incorrect.

EXCESSIVE BAD BLKS IN FREE LIST (CONTINUE)

The free-block list contains more than a tolerable number (usually 10) of blocks with a value less than the first data block in the file system or greater than the last block in the file system. See Sections 4.1.2 and 4.2.4.

Possible responses to the CONTINUE prompt are:

YES	ignore the rest of the free-block list and continue the execution of *fsck*. This error condition will always invoke the BAD BLKS IN FREE LIST error condition in Phase 5.
NO	terminate the program.

EXCESSIVE DUP BLKS IN FREE LIST (CONTINUE)

The free-block list contains more than a tolerable number (usually 10) of blocks claimed by inodes or earlier parts of the free-block list. See Sections 4.1.2 and 4.2.3.

Possible responses to the CONTINUE prompt are:

YES	ignore the rest of the free-block list and continue the execution of *fsck*. This error condition will always invoke the DUP BLKS IN FREE LIST error condition in Phase 5.
NO	terminate the program.

BAD FREEBLK COUNT

The count of free blocks in a free-list block is greater than 50 or less than zero. This error condition will always invoke the BAD FREE LIST condition in Phase 5. See Section 4.1.2.

X BAD BLKS IN FREE LIST

X blocks in the free-block list have a block number lower than the first data block in the file system or greater than the last block in the file system. This error condition will always invoke the BAD FREE LIST condition in Phase 5. See Sections 4.1.2 and 4.2.4.

X DUP BLKS IN FREE LIST

X blocks claimed by inodes or earlier parts of the free-list block were found in the free-block list. This error condition will always invoke the BAD FREE LIST condition in Phase 5. See Sections 4.1.2 and 4.2.3.

X BLK(S) MISSING

X blocks unused by the file system were not found in the free-block list. This error condition will always invoke the BAD FREE LIST condition in Phase 5. See Section 4.1.2.

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. See Section 4.1.3.

Possible responses to the FIX prompt are:

YES	replace the count in the super-block by the actual count.
NO	ignore this error condition.

BAD FREE LIST (SALVAGE)

Phase 5 has found bad blocks in the free-block list, duplicate blocks in the free-block list, or blocks missing from the file system. See Sections 4.1.2, 4.2.3, and 4.2.4. If the **−q** option is specified, the free-block list will be salvaged automatically.

Possible responses to the SALVAGE prompt are:

YES	replace the actual free-block list with a new free-block list. The new free-block list will be ordered to reduce time spent by the disk waiting for the disk to rotate into position.
NO	ignore this error condition.

9. PHASE 6: SALVAGE FREE LIST

This phase concerns itself with the free-block list reconstruction. This section lists error conditions resulting from the blocks-to-skip and blocks-per-cylinder values.

Default free-block list spacing assumed

This is an advisory message indicating the blocks-to-skip is greater than the blocks-per-cylinder, the blocks-to-skip is less than one, the blocks-per-cylinder is less than one, or the blocks-per-cylinder is greater than 500. The default values of 9 blocks-to-skip and 400 blocks-per-cylinder are used. See the *fsck*(1M) manual entry for further details.

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.

X files Y blocks Z free

This is an advisory message indicating that the file system checked contained X files using Y blocks leaving Z blocks free in the file system.

******* BOOT UNIX (NO SYNC!) *******

This is an advisory message indicating that a mounted file system or 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.

******* 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.

INDEX OF MESSAGES
(Alphabetically within each section)

INITIALIZATION

PHASE 1: CHECK BLOCKS AND SIZES

PHASE 1B: RESCAN FOR MORE DUPS

PHASE 2: CHECK PATH-NAMES

PHASE 3: CHECK CONNECTIVITY

PHASE 4: CHECK REFERENCE COUNTS

PHASE 5: CHECK FREE LIST

PHASE 6: SALVAGE FREE LIST

CLEANUP

The UNIX Accounting System

H. S. McCreary
A. G. Petruccelli

Bell Laboratories
Piscataway, New Jersey 08854

ABSTRACT

The UNIX† Accounting System provides methods to collect per-process resource utilization data, to record connect sessions, to monitor disk utilization, and to charge fees to specific logins. A set of C programs and shell procedures is provided to reduce this accounting data into summary files and reports. This memorandum describes the structure, implementation, and management of this accounting system, as well as a discussion of the reports generated and the meaning of the columnar data.

1. INTRODUCTION

The UNIX Accounting System was originally designed by John Mashey. Several modifications and additions have been made to make the system easier to manage and to make it less susceptible to corrupted data or system errors. The following list is a synopsis of the actions of the accounting system:

- At process termination the UNIX Kernel writes one record per process in **/usr/adm/pacct** in the form of *acct.h*.[1]
- The *login* and *init* programs record connect sessions by writing records into **/usr/adm/wtmp**. Date changes, reboots, and shutdowns are also recorded in this file.
- The disk utilization program *acctdusg*, breaks down disk usage by login.
- Fees for file restores, etc, can be charged to specific logins with the *chargefee* shell procedure.
- Each day the *runacct* shell procedure is executed via *cron* to reduce accounting data, produce summary files and reports.[2]
- The *monacct* procedure can be executed on a monthly or fiscal period basis. It saves and restarts summary files, generates a report, and cleans up the **sum** directory. These saved summary files could be used to charge users for UNIX usage.

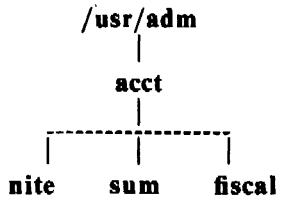
2. FILES AND DIRECTORIES

The **/usr/lib/acct** directory contains all of the C programs and shell procedures necessary to run the accounting system. The **adm** login (currently user ID of 4) is used by the accounting system and has the following directory structure:

---

† UNIX is a trademark of Bell Laboratories.

1. See Attachment 2 for a description of data files.
2. See Attachment 3 for a sample report.



The **/usr/adm** directory contains the active data collection files.[3] The **nite** directory contains files that are re-used daily by the *runacct* procedure. The **sum** directory contains the cumulative summary files updated by *runacct*. The **fiscal** directory contains periodic summary files created by *monacct*.

3. DAILY OPERATION

When UNIX is switched into multi-user mode, **/usr/lib/acct/startup** is executed which does the following:

1. The *acctwtmp* program adds a "boot" record to **/usr/adm/wtmp**. This record is signified by using the system name as the login name in the **wtmp** record.
2. Process accounting is started via *turnacct*. *Turnacct on* executes the *accton* program with the argument **/usr/adm/pacct**.
3. The *remove* shell procedure is executed to clean up the saved **pacct** and **wtmp** files left in the **sum** directory by *runacct*.

The *ckpacct* procedure is run via *cron* every hour of the day to check the size of **/usr/adm/pacct**. If the file grows past 1000 blocks (default), *turnacct switch* is executed. While *ckpacct* is not absolutely necessary, the advantage of having several smaller **pacct** files becomes apparent when trying to restart *runacct* after a failure processing these records.

The *chargefee* program can be used to bill users for file restores, etc. It adds records to **/usr/adm/fee** which are picked up and processed by the next execution of *runacct* and merged into the total accounting records.

Runacct is executed via *cron* each night. It processes the active accounting files, **/usr/adm/pacct***?*, **/usr/adm/wtmp**, **/usr/adm/acct/nite/disktacct**, and **/usr/adm/fee**. It produces command summaries and usage summaries by login.

When the system is shut down using *shutdown*, the *shutacct* shell procedure is executed. It writes a shutdown reason record into **/usr/adm/wtmp** and turns process accounting off.

After the first re-boot each morning, the computer operator should execute **/usr/lib/acct/prdaily** to print the previous day's accounting report.

4. SETTING UP THE ACCOUNTING SYSTEM

In order to automate the operation of this accounting system, several things need to be done:

1. If not already present, add this line to the **/etc/rc** file in the state 2 section:

   /bin/su − adm −c /usr/lib/acct/startup

2. If not already present, add this line to */etc/shutdown* to turn off the accounting before the system is brought down:

---

3. For a complete explanation of the files used by the accounting system, see Attachment 1.

/usr/lib/acct/shutacct

3. For most installations, the following three entries should be made in **/usr/lib/crontab** so that *cron* will automatically run the daily accounting:

```
0 4 * * 1-6 /bin/su - adm -c "/usr/lib/acct/runacct 2> /usr/adm/acct/nite/fd2log"
0 2 * * 4 /usr/lib/acct/dodisk
5 * * * * /bin/su - adm -c "/usr/lib/acct/ckpacct"
```

Note that *dodisk* is invoked with super-user privileges of **root** so that directory searching is not road blocked.

To facilitate monthly merging of accounting data, the following entry in **crontab** will allow *monacct* to clean up all daily reports and daily total accounting files and deposit one monthly total report and one monthly total accounting file in the **fiscal** directory:

```
15 5 1 * * /bin/su - adm -c /usr/lib/acct/monacct
```

The above entry takes advantage of the default action of *monacct* that uses the current month's date as the suffix for the file names. Notice that the entry is executed at such a time as to allow *runacct* sufficient time to complete. This will, on the first day of each month, create monthly accounting files with the entire month's data.

4. The **PATH** shell variable should be set in **/usr/adm/.profile** to:

```
PATH=/usr/lib/acct:/bin:/usr/bin
```

5. RUNACCT

Runacct is the main daily accounting shell procedure. It is normally initiated via *cron* during non-prime time hours. *Runacct* processes connect, fee, disk, and process accounting files. It also prepares daily and cumulative summary files for use by *prdaily* or for billing purposes. The following files produced by *runacct* are of particular interest:

nite/lineuse	Produced by *acctcon1*, which reads the **wtmp** file, and produces usage statistics for each terminal line on the system. This report is especially useful for detecting bad lines. If the ratio between the number of logoffs to logins exceeds about 3/1, there is a good possibility that the line is failing.
nite/daytacct	This file is the total accounting file for the previous day in *tacct.h* format.
sum/tacct	This file is the accumulation of each day's **nite/daytacct**, which can be used for billing purposes. It is restarted each month or fiscal by the *monacct* procedure.
sum/daycms	Produced by the *acctcms* program, it contains the daily command summary. The ASCII version of this file is **nite/daycms**.
sum/cms	The accumulation of each day's command summaries. It is restarted by the execution of *monacct*. The ASCII version is **nite/cms**.
sum/loginlog	Produced by the *lastlogin* shell procedure, it maintains a record of the last time each login was used.
sum/rprt.MMDD	Each execution of *runacct* saves a copy of the output of *prdaily*.

Runacct takes care not to damage files in the event of errors. A series of protection mechanisms are used that attempt to recognize an error, provide intelligent diagnostics, and terminate processing in such a way that *runacct* can be restarted with minimal intervention. It records its

progress by writing descriptive messages into the file **active.**[4] All diagnostic output during the execution of *runacct* is written into **fd2log**. To prevent multiple invocations, in the event of two crons or other problems, *runacct* will complain if the files **lock** and **lock1** exist when invoked. The **lastdate** file contains the month and day *runacct* was last invoked, and is used to prevent more than one execution per day. If *runacct* detects an error, a message is written to **/dev/console**, mail is sent to **root** and **adm**, the locks are removed, diagnostic files are saved, and execution is terminated.

In order to allow *runacct* to be restartable, processing is broken down into separate reentrant states. This is accomplished by using a **case** statement inside an endless **while** loop. Each state is one case of the **case** statement. A file is used to remember the last state completed. When each state completes, **statefile** is updated to reflect the next state. In the next loop through the **while**, **statefile** is read and the **case** falls through to the next state. When *runacct* reaches the CLEANUP state, it removes the locks and terminates. *States* are executed as follows:

SETUP	The command **turnacct switch** is executed. The process accounting files, **/usr/adm/pacct***?*, are moved to **/usr/adm/Spacct***?.MMDD*. The **/usr/adm/wtmp** file is moved to **/usr/adm/acct/nite/wtmp.***MMDD* with the current time added on the end.
WTMPFIX	The **wtmp** file in the **nite** directory is checked for correctness by the *wtmpfix* program. Some date changes will cause *acctcon1* to fail, so *wtmpfix* attempts to adjust the time stamps in the **wtmp** file if a date change record appears.
CONNECT1	Connect session records are written to **ctmp** in the form of *ctmp.h*. The **lineuse** file is created, and the **reboots** file is created showing all of the boot records found in the **wtmp** file.
CONNECT2	**Ctmp** is converted to **ctacct.***MMDD*, the connect accounting records.[5]
PROCESS	The *acctprc1* and *acctprc2* programs are used to convert the process accounting files, **/usr/adm/Spacct***?.MMDD*, into total accounting records in **ptacct***?.MMDD*. The **Spacct** and **ptacct** files are correlated by number so that if *runacct* fails, the unnecessary reprocessing of **Spacct** files will not occur. One precaution should be noted; when restarting *runacct* in this state, remove the last ptacct file because it will not be complete.
MERGE	Merge the process accounting records with the connect accounting records to form **daytacct**.
FEES	Merge in any ASCII *tacct* records from the file **fee** into **daytacct**.
DISK	On the day after the *sdisk* procedure runs, merge **disktacct** with **daytacct**.
MERGETACCT	Merge **daytacct** with **sum/tacct**, the cumulative total accounting file. Each day, **daytacct** is saved in **sum/tacct***MMDD*, so that **sum/tacct** can be recreated in the event it becomes corrupted or lost.
CMS	Merge in today's command summary with the cumulative command summary file **sum/cms**. Produce ASCII and internal format command summary files.

4. Files used by *runacct* are assumed to be in the **nite** directory unless otherwise noted.

5. Accounting records are in *tacct.h* format.

USEREXIT — Any installation dependent (local) accounting programs can be included here.

CLEANUP — Clean up temporary files, run *prdaily* and save its output in **sum/rprt***MMDD*, remove the locks, then exit.

6. RECOVERING FROM FAILURE

The *runacct* procedure can fail for a variety of reasons; usually due to a system crash, **/usr** running out of space, or a corrupted **wtmp** file. If the **active***MMDD* file exists, check it first for error messages. If the **active** file and lock files exist, check **fd2log** for any mysterious messages. The following are error messages produced by *runacct*, and the recommended recovery actions:

ERROR: locks found, run aborted
The files **lock** and **lock1** were found. These files must be removed before *runacct* can restart.

ERROR: acctg already run for *date* : check /usr/adm/acct/nite/lastdate
The date in **lastdate** and today's date are the same. Remove **lastdate**.

ERROR: turnacct switch returned rc=*?*
Check the integrity of *turnacct* and *accton*. The *accton* program must be owned by **root**, and have the setuid bit set.

ERROR: Spacct*?*.*MMDD* already exists
file setups probably already run
Check status of files, then run setups manually.

ERROR: /usr/adm/acct/nite/wtmp.*MMDD* already exists, run setup manually
Self-explanatory.

ERROR: wtmpfix errors see /usr/adm/acct/nite/wtmperror
Wtmpfix detected a corrupted **wtmp** file. Use *fwtmp* to correct the corrupted file.

ERROR: connect acctg failed: check /usr/adm/acct/nite/log
The *acctcon1* program encountered a bad **wtmp** file. Use *fwtmp* to correct the bad file.

ERROR: Invalid state, check /usr/adm/acct/nite/active
The file, **statefile**, is probably corrupted. Check **statefile** and read **active** before restarting.

7. RESTARTING RUNACCT

Runacct called without arguments assumes that this is the first invocation of the day. The argument *MMDD* is necessary if *runacct* is being restarted, and specifies the month and day for which *runacct* will rerun the accounting. The entry point for processing is based on the contents of **statefile**. To override **statefile**, include the desired *state* on the command line. For example:

To start *runacct*:

```
nohup runacct  2>  /usr/adm/acct/nite/fd2log&
```

To restart *runacct*:

```
nohup runacct 0601  2>  /usr/adm/acct/nite/fd2log&
```

To restart *runacct* at a specific state:

```
nohup runacct 0601 WTMPFIX  2>  /usr/adm/acct/nite/fd2log&
```

8. FIXING CORRUPTED FILES

Unfortunately, this accounting system is not entirely fool proof. Occasionally a file will become corrupted, or lost. Some of the files can simply be ignored or restored from the file-save backup, but others must be fixed to maintain the integrity of the accounting system.

8.1 Fixing WTMP Errors

The **wtmp** files seem to cause the most problems in the day to day operation of the accounting system. When the date is changed when UNIX is in multi-user mode, a set of date change records is written into **/usr/adm/wtmp**. The *wtmpfix* program is designed to adjust the time stamps in the **wtmp** records when a date change is encountered. Some combinations of date changes and reboots, however, will slip through *wtmpfix* and cause *acctcon1* to fail. The following steps show how to patch up a **wtmp** file.

```
cd /usr/adm/acct/nite
fwtmp < wtmp.MMDD > xwtmp
ed xwtmp
    delete corrupted records or
    delete all records from the beginning up to the date change
fwtmp -ic < xwtmp > wtmp.MMDD
```

If the **wtmp** file is beyond repair, create a null **wtmp** file. This will prevent any charging of connect time. *Acctprc1* won't be able to determine which login owned a particular process, but it will be charged to the login that is first in the password file for that userid.

8.2 Fixing TACCT Errors

If the installation is using the accounting system to charge users for system resources, the integrity of **sum/tacct** is quite important. Occasionally, mysterious **tacct** records will appear with negative numbers, duplicate user IDs, or a user ID of 65535. First check **sum/tacctprev** with *prtacct*. If it looks all right, the latest **sum/tacct.***MMDD* should be patched up, then **sum/tacct** recreated. A simple patchup procedure would be:

```
cd /usr/adm/acct/sum
acctmerg -v < tacct.MMDD > xtacct
ed xtacct
    remove the bad records
    write duplicate uid records to another file
acctmerg -i < xtacct > tacct.MMDD
acctmerg tacctprev < tacct.MMDD > tacct
```

Remember that the *monacct* procedure removes all the **tacct.***MMDD* files; therefore, **sum/tacct** can be recreated by merging these files together.

9. UPDATING PNPSPLIT

The *pnpsplit* subroutine is used by *acctcon1* and *acctprc1* to determine the difference between prime and non-prime time. Prime time is defaulted from 9 a.m. to 5 p.m. Monday through Friday. Non-prime time is considered to be all other hours and the entire day for those days listed in the **holidays** structure in **pnpsplit.c**. The holidays listed are accurate for Bell Laboratories, New Jersey locations for the year the operating system was released. Every year on the day after Christmas (the last holiday of the calendar year), the following message will be printed on the system console terminal and appear in **log**:

```
*** RECOMPILE pnpsplit WITH NEW HOLIDAYS ***
```

This message will continue to be sent each time the accounting is run until *pnpsplit*, *acctcon1*, and *acctprc1* are recompiled. The following steps should be taken to successfully recompile these programs:

1. Edit **pnpsplit.c** to change the **thisyear** variable to the new year. Update the **holidays** structure to reflect the new holidays. The numeric entry in the structure is the day of the year, less one. For example, New Year's Day (January 1) is entered as 0. **Pnpsplit.c** is in **/usr/src/cmd/acct/lib**.

2. Update the accounting library *a.a* and recompile *acctprc1*, and *acctcon1* by:

   super-user to root
   ARGS="acctcon1 acctprc1" /usr/src/:mkcmd acct

10. DAILY REPORTS

Runacct generates 5 basic reports upon each invocation. A sample of these reports are shown in Attachment 3. They cover the areas of connect accounting, usage by person on a daily basis, command usage reported by daily and monthly totals, and a report of the last time users were logged in.

The following paragraphs describe the reports and the meanings of their tabulated data.

10.1 Daily Report

In the first part of the report, the *from/to* banner should alert you to the period reported on. The times are the time the last accounting report was generated until the time the current accounting report was generated. It is followed by a log of system reboots, shutdowns, power fail recoveries, and any other record dumped into **/usr/adm/wtmp** by the *acctwtmp* program (see *acct*(1M)).

The second part of the report is a breakdown of line utilization. The **TOTAL DURATION** tells how long the system was in multi-user state (able to be accessed through the terminal lines). The columns are:

LINE The terminal line or access port.

MINUTES The total number of minutes that line was in use during the accounting period.

PERCENT The total number of MINUTES the line was in use divided into the TOTAL DURATION.

SESS The number of times this port was accessed for a *login*(1) session.

ON This column does not have much meaning anymore. It used to give the number of times that the port was used to log a user on, but because *login*(1) can no longer be executed explicitly to log a new user in, this column should be identical with SESS.

OFF This column reflects not just the number of times a user logged off, but also any interrupts that occur on that line. Generally, interrupts occur on a port when the *getty*(8) is first invoked when the system is brought to multi-user state. These interrupts occur at a rate of about two per event; therefore it is not uncommon to see in excess of twice the amount of OFF than in ON or SESS. Where this column does come into play is when the # OFF exceeds the # ON by a large factor. This usually indicates that the multiplexor, modem or cable is going bad, or there is a bad connection somewhere. The most common cause of this is an unconnected cable dangling from the multiplexor.

During real time, **/usr/adm/wtmp** should be monitored as this is the file that the connect accounting is geared off of. If it grows rapidly, execute *acctcon1* to see which *tty* line is the most noisy. If the interrupting is occurring at a furious rate, you'll be able to feel the effect on general system performance.

10.2 Daily Usage Report

This report gives a by-user breakdown of system resource utilization. Its data consists of:

UID — The user ID.

LOGIN NAME — The login name of the user; there can be more than one login name for a single user ID, this identifies which one.

CPU (MINS) — This represents the amount of time the user's process used the central processing unit. This category is broken down into PRIME and NPRIME (non-prime) utilization. The accounting system's idea of this breakdown is located in the accounting library function *prpsplit* where the **holidays** array, which also determines non-prime time, is also defined. As delivered, prime time is defined to be 0900-1700 hours. The **holidays** array is correct for New Jersey locations of Bell Laboratories for the year of the release.

KCORE-MINS — This represents a cummulative measure of the amount of memory a process uses while running. The amount shown reflects kilo-byte segments of memory used per minute. This measurement is also broken down into PRIME and NPRIME amounts.

CONNECT (MINS) — This identifies "Real Time" used. What this column really identifies, is the amount of time that a user was logged into the system. If this time is rather high and the later column called # OF PROCS is low, this user is what is called a "line hog." That is, this person logs in first thing in the morning and doesn't hardly touch the terminal the rest of the day. Watch out for this kind of critter. This column is also subdivided into PRIME and NPRIME utilization.

DISK BLOCKS — When the disk accounting programs have been run, their output is merged into the total accounting record (**tacct.h**) and shows up in this column. This disk accounting is accomplished by the program *acctdusg*.

OF PROCS — This column reflects the number of processes that was invoked by the user. This is a good column to watch for large numbers indicating that a user may have a shell procedure that runs amuck. The most common example of this is for a **crontab** entry to try to execute a user's **.profile** via **su**— that unfortunately prompts for a terminal type and sits in an endless loop trying to read from the terminal (there isn't one when *cron* is executing a process). Preventive coding is encouraged in the **.profile**.

OF SESS — This is how many times the user logged onto the system.

DISK SAMPLES — This indicates how many times the disk accounting was run to obtain the average number of DISK BLOCKS listed earlier.

FEE — A much often unused field in the total accounting record, the FEE represents the total accumulation of *widgets* charged against the user by the *chargefee* shell procedure (see *acctsh*(1M)). The *chargefee* procedure is used to levy charges against a user for special services performed such as file restores, tape manipulation by operators, etc.

10.3 Daily Command and Monthly Total Command Summaries

These two reports are virtually the same except that the Daily Command Summary only reports on the current accounting period, while the Monthly Total Command Summary tells the story for the start of the fiscal period to the current date. In other words, the monthly report reflects the data accumulated since the last invocation of *monacct*.

The data included in these reports give an administrator an idea as to the heaviest used commands, and based on those commands' characteristics of system resource utilization, a hint as to what to weigh more heavily when system tuning.

These reports are sorted by TOTAL KCOREMIN which is an arbitrary yardstick, but often a good one for calculating "drain" caused on a system.

COMMAND NAME	This is the name of the command. Unfortunately, all shell procedures are lumped together under the name **sh** because only object modules are reported by the process accounting system. The administrator should monitor the frequency of programs called *a.out* or *core* or any other name that doesn't seem quite right. Often people like to work on their favorite version of *backgammon* only they don't want everyone to know about it. *Accton* is also a good tool to use for determining who executed a suspiciously named command and also if super-user privileges were used.
NUMBER CMDS	This is the total number of invocations of this particular command.
TOTAL KCOREMIN	The total cummulative measurement of the amount of kilo-byte segments of memory used by a process per minute of run time.
TOTAL CPU-MIN	The total processing time this program has accumulated.
TOTAL REAL-MIN	The total real time (wall-clock) minutes this program has accumulated. This total is the actual "waited for" time as opposed to kicking off a process in the background.
MEAN SIZE-K	This is the mean of the TOTAL KCOREMIN over the number of invocations reflected by NUMBER CMDS.
MEAN CPU-MIN	This is the mean derived between the NUMBER CMDS and TOTAL CPU-MIN.
HOG FACTOR	This is a relative measurement of the ratio of system availability to system utilization. It is computed by the formula (total CPU time) / (elapsed time) This gives a relative measure of the total available CPU time consumed by the process during its execution.
CHARS TRNSFD	This column, which may go negative, is a total count of the number of characters pushed around by the *read*(2) and *write*(2) system calls.
BLOCKS READ	A total count of the physical block reads and writes that a process performed.

10.4 Last Login

This report simply gives the date when a particular login was last used. This could be a good source for finding likely candidates for the tape archives or getting rid of unused logins and login directories.

11. SUMMARY

The UNIX Accounting System was designed from a UNIX system administrator's point of view. Every possible precaution has been taken to ensure that the system will run smoothly and without error. It is important to become familiar with the C programs and shell procedures. The manual entries should be studied, and it is advisable to keep a printed copy of the shell procedures handy. This accounting system should be easy to maintain, provide valuable information for the administrator, and provide accurate breakdowns of the usage of system resources for charging purposes.

Attachment 1

Files in the /usr/adm directory:

diskdiag	diagnostic output during the execution of disk accounting programs
dtmp	output from the *acctdusg* program
fee	output from the *chargefee* program, ASCII tacct records
pacct	active process accounting file
pacct?	process accounting files switched via *turnacct*
Spacct?.MMDD	process accounting files for *MMDD* during execution of *runacct*
wtmp	active wtmp file for recording connect sessions

Files in the /usr/adm/acct/nite directory:

active	used by *runacct* to record progress and print warning and error messages; **active***MMDD* same as **active** after *runacct* detects an error
cms	ASCII total command summary used by *prdaily*
ctacct.MMDD	connect accounting records in *tacct.h* format
ctmp	output of *acctcon1* program, connect session records in *ctmp.h* format
daycms	ASCII daily command summary used by *prdaily*
daytacct	total accounting records for one day in *tacct.h* format
disktacct	disk accounting records in *tacct.h* format, created by *dodisk* procedure
fd2log	diagnostic output during execution of runacct (see *cron* entry)
lastdate	last day *runacct* executed in *date* +*%m%d* format
lock lock1	used to control serial use of *runacct*
lineuse	tty line usage report used by *prdaily*
log	diagnostic output from *acctcon1*
logMMDD	same as **log** after *runacct* detects an error
reboots	contains beginning and ending dates from **wtmp**, and a listing of reboots
statefile	used to record current state during execution of *runacct*
tmpwtmp	wtmp file corrected by *wtmpfix*
wtmperror	place for *wtmpfix* error messages
wtmperrorMMDD	same as **wtmperror** after *runacct* detects an error
wtmp.MMDD	previous day's wtmp file

Files in the /usr/adm/acct/sum directory:

cms	total command summary file for current fiscal in internal summary format
cmsprev	command summary file without latest update
daycms	command summary file for yesterday in internal summary format
loginlog	created by *lastlogin*
pacct.MMDD	concatenated version of all pacct files for *MMDD*, removed after reboot by *remove* procedure
rprt.MMDD	saved output of *prdaily* program
tacct	cumulative total accounting file for current fiscal
tacctprev	same as **tacct** without latest update
tacct.MMDD	total accounting file for *MMDD*
wtmp.MMDD	saved copy of wtmp file for *MMDD*, removed after reboot by *remove* procedure

Files in the /usr/adm/acct/fiscal directory:

cms?	total command summary file for fiscal *?* in internal summary format
fiscrpt?	report similar to *prdaily* for fiscal *?*
tacct?	total accounting file for fiscal *?*

Attachment 2

Format of wtmp files (utmp.h):

```
/*
 * Format of /etc/utmp and /usr/adm/wtmp
 */
struct utmp {
        char    ut_line[8];             /* tty name */
        char    ut_name[8];             /* user id */
        long    ut_time;                /* time on */
};
```

Definitions (acctdef.h):

```
/*
 *      defines, typedefs, etc. used by acct programs
 *
 *      acct only typedefs
 */
typedef unsigned short  uid_t;

#define LSZ     8       /* sizeof line name */
#define NSZ     8       /* sizeof login name */
#define P       0       /* prime time */
#define NP      1       /* nonprime time */

/*
 *      limits which may have to be increased if systems get larger
 */
#define SSIZE  1000     /* max number of sessions in 1 acct run */
#define TSIZE  100      /* max number of line names in 1 acct run */
#define USIZE 500       /* max number of distinct login names in 1 acct run */

#define EQN(s1, s2)   (strncmp(s1, s2, sizeof(s1)) == 0)
#define CPYN(s1, s2)  strncpy(s1, s2, sizeof(s1))

#define SECS(tics)      ((double) tics)/60.
#define MINS(secs)      ((double) secs)/60.
#define MINT(tics)      ((double) tics)/3600.
#ifdef pdp11
#define KCORE(clicks)           ((double) clicks/16)
#endif
#ifdef vax
#define KCORE(clicks)           ((double) clicks/2)
#endif
#define SECSINDAY   86400L
```

Format of pacct files (acct.h):

```
/*
 * Accounting structures
 */

typedef ushort comp_t;          /* "floating point" */
                /* 13-bit fraction, 3-bit exponent  */

struct  acct
{
        char    ac_flag;        /* Accounting flag */
        char    ac_stat;        /* Exit status */
        ushort  ac_uid;         /* Accounting user ID */
        ushort  ac_gid;         /* Accounting group ID */
        dev_t   ac_tty;         /* control typewriter */
        time_t  ac_btime;       /* Beginning time */
        comp_t  ac_utime;       /* acctng user time in clock ticks */
        comp_t  ac_stime;       /* acctng system time in clock ticks */
        comp_t  ac_etime;       /* acctng elapsed time in clock ticks */
        comp_t  ac_mem;         /* memory usage */
        comp_t  ac_io;          /* chars transferred */
        comp_t  ac_rw;          /* blocks read or written */
        char    ac_comm[8];     /* command name */
};

extern  struct  acct    acctbuf;
extern  struct  inode   *acctp; /* inode of accounting file */

#define AFORK           01      /* has executed fork, but no exec */
#define ASU             02      /* used super-user privileges */
#define ACCTF           0300    /* record type: 00 = acct */
```

Format of tacct files (tacct.h):

```
/*
 *      total accounting (for acct period), also for day
 */

struct  tacct   {
        uid_t           ta_uid;         /* userid */
        char            ta_name[8];     /* login name */
        float           ta_cpu[2];      /* cum. cpu time, p/np (mins) */
        float           ta_kcore[2];    /* cum kcore-minutes, p/np */
        float           ta_con[2];      /* cum. connect time, p/np, mins */
        float           ta_du;          /* cum. disk usage */
        long            ta_pc;          /* count of processes */
        unsigned short  ta_sc;          /* count of login sessions */
        unsigned short  ta_dc;          /* count of disk samples */
        unsigned short  ta_fee;         /* fee for special services */
};
```

Format of ctmp file (ctmp.h):

```
/*
 *      connect time record (various intermediate files)
 */
struct ctmp {
        dev_t   ct_tty;                 /* major minor */
        uid_t   ct_uid;                 /* userid */
        char    ct_name[8];             /* login name */
        long    ct_con[2];              /* connect time (p/np) secs */
        time_t  ct_start;               /* session start time */
};
```

Attachment 3

Jun 8 04:14 1979 DAILY REPORT FOR pwba Page 1

from Thu Jun 7 06:00:48 1979
to Fri Jun 8 04:00:28 1979
2 shutdown
2 pwba

TOTAL DURATION IS 1320 MINUTES

LINE	MINUTES	PERCENT	# SESS	# ON	# OFF
tty04	479	36	9	9	30
tty47	341	26	4	4	33
tty44	298	23	3	3	29
tty46	336	25	9	9	33
console	1100	83	14	14	21
tty05	448	34	3	3	22
tty06	439	33	9	9	31
tty07	421	32	6	6	24
tty42	53	4	5	5	20
tty09	385	29	11	11	33
tty10	336	25	10	10	31
tty08	464	35	2	2	19
tty26	544	41	6	6	24
tty12	252	19	5	5	25
tty13	258	20	3	3	21
tty14	156	12	6	6	26
tty17	145	11	1	1	16
tty18	39	3	5	5	24
tty15	228	17	5	5	25
tty25	704	53	6	6	25
tty21	0	0	0	0	16
tty19	10	1	1	1	17
tty20	25	2	2	2	18
tty22	0	0	0	0	15
tty23	0	0	0	0	15
tty24	0	0	0	0	16
tty27	481	36	3	3	20
tty28	426	32	5	5	24
tty29	302	23	6	6	25
tty30	257	20	11	11	28
tty40	380	29	5	5	21
tty41	343	26	3	3	21
tty45	0	0	0	0	15
tty11	365	28	7	7	25
tty43	3	0	1	1	17
tty16	213	16	3	3	20
tty31	250	19	4	4	18
tty02	62	5	1	1	3
TOTALS	10544	--	174	174	846

Jun 8 04:14 1979 DAILY USAGE REPORT FOR pwba Page 1

UID	LOGIN NAME	CPU (MINS) PRIME	CPU (MINS) NPRIME	KCORE-MINS PRIME	KCORE-MINS NPRIME	CONNECT (MINS) PRIME	CONNECT (MINS) NPRIME	DISK BLOCKS	# OF PROCS	# OF SESS	# DISK SAMPLES	FEE
0	TOTAL	388	103	12414	2934	9251	1056	0	16164	174	0	0
0	root	47	41	1003	924	67	30	0	2360	8	0	0
4	adm	7	19	48	652	0	0	0	842	0	0	0
19	games	0	0	4	0	0	0	0	28	0	0	0
22	mhb	0	0	1	1	1	1	0	14	2	0	0
37	abs	0	0	4	0	0	0	0	3	0	0	0
37	absjrk	14	0	284	0	423	0	0	1588	4	0	0
68	rje	3	3	24	21	0	0	0	179	0	0	0
71	?	0	0	0	0	0	0	0	12	0	0	0
150	jac	7	0	156	5	281	2	0	510	13	0	0
173	?	0	0	0	0	0	0	0	16	0	0	0
180	?	0	0	0	0	0	0	0	4	0	0	0
185	?	0	0	0	0	0	0	0	2	0	0	0
217	denise	0	0	2	0	31	0	0	32	3	0	0
217	kof	0	0	2	0	1	0	0	7	1	0	0
219	?	0	0	0	0	0	0	0	12	0	0	0
1001	hsm	5	0	189	0	179	0	0	92	2	0	0
2001	systst	0	1	5	28	476	64	0	99	5	0	0
2002	mfp	1	0	7	5	270	62	0	93	3	0	0
2003	als	1	0	23	0	100	0	0	99	3	0	0
2005	eric	0	0	3	0	13	0	0	21	1	0	0
2006	hoot	0	0	2	0	16	0	0	8	1	0	0
2009	agp	47	0	2040	0	444	0	0	492	2	0	0
2009	fsrep1	2	0	60	0	36	0	0	95	1	0	0
2011	pdw	0	0	1	0	4	0	0	11	1	0	0
2012	pwbst	0	0	1	0	28	0	0	9	1	0	0
2014	cath	0	0	1	0	1	0	0	7	1	0	0
2022	rem	32	1	1227	91	576	4	0	226	3	0	0
2025	fld	55	23	2176	862	336	98	0	750	7	0	0
2027	krb	14	2	365	51	547	24	0	372	8	0	0
2028	text	0	0	1	0	3	0	0	13	1	0	0
2030	arf	8	0	288	0	317	0	0	315	3	0	0
2031	dp	12	0	480	3	459	6	0	220	6	0	0
2032	graf	2	0	49	0	23	0	0	118	1	0	0
2033	ecp	3	0	74	0	355	0	0	115	4	0	0
2040	leap	15	0	308	0	513	1	0	505	2	0	0
2041	dan	3	0	93	3	149	2	0	117	8	0	0
2051	ds52	2	2	19	40	375	601	0	611	8	0	0
2055	nuucp	0	0	15	9	17	1	0	10	3	0	0
2057	ech	1	0	28	0	63	0	0	68	2	0	0
2061	jcw	4	3	99	70	37	34	0	869	4	0	0
2064	mjr	18	0	443	0	176	0	0	2065	3	0	0
2065	rrr	0	0	6	0	7	0	0	23	1	0	0
2068	trc	0	0	7	0	10	0	0	29	1	0	0
2075	herb	29	0	1178	1	384	2	0	249	5	0	0
2086	paul	1	0	14	0	152	0	0	28	1	0	0
2087	pris	0	0	0	10	0	2	0	13	1	0	0
2111	pwbcs	2	3	60	85	64	86	0	185	4	0	0
2116	rbj	1	0	16	0	408	0	0	222	1	0	0
2121	teach	0	0	3	0	53	0	0	50	2	0	0
2123	msb	0	0	3	0	5	0	0	24	1	0	0
2124	rnt	2	0	42	0	66	0	0	260	3	0	0
2126	dal	0	0	5	0	121	0	0	17	1	0	0
2127	m2	15	0	495	11	390	2	0	602	10	0	0

Jun 8 04:14 1979 DAILY USAGE REPORT FOR pwba Page 2

2128	jel	14	0	492	9	422	14	0	523	8	0	0
2130	s1	0	0	5	1	16	0	0	42	2	0	0
2130	s3	0	0	0	0	0	2	0	9	1	0	0
2135	jfn	0	1	0	12	0	11	0	33	2	0	0
2136	m2class	0	0	5	0	2	0	0	18	1	0	0
2140	star	4	0	213	12	90	3	0	170	7	0	0
2141	reg	5	0	245	25	470	4	0	181	1	0	0
2199	llc	0	0	1	0	10	0	0	7	1	0	0
2999	stock	0	0	1	0	1	0	0	17	1	0	0
3001	whm	5	0	93	0	253	0	0	414	3	0	0
3332	vjf	0	0	4	0	8	0	0	39	1	0	0

Jun 8 04:07 1979 DAILY COMMAND SUMMARY Page 1

COMMAND NAME	NUMBER CMDS	TOTAL KCOREMIN	TOTAL CPU-MIN	TOTAL REAL-MIN	MEAN SIZE-K	MEAN CPU-MIN	HOG FACTOR	CHARS TRNSFD	BLOCKS READ
TOTALS	16164	15332.89	490.72	37463.98	31.25	0.03	0.01	322183844	1097670
nroff	119	3958.68	93.21	569.83	42.47	0.78	0.16	67070052	130284
troff	26	2483.38	51.63	342.70	48.10	1.99	0.15	37869304	48989
xnroff	20	732.03	16.74	111.05	43.73	0.84	0.15	13885248	22659
a.out	31	623.53	10.52	142.77	59.26	0.34	0.07	382435	2758
egrep	185	574.83	13.96	34.53	41.18	0.08	0.40	170625	8249
m2find	232	555.79	9.93	155.11	55.96	0.04	0.06	6155937	30994
c1	150	519.04	13.57	48.89	38.25	0.09	0.28	4285724	16032
c0	165	413.10	9.19	35.16	44.93	0.06	0.26	3827309	12170
m2edit	33	340.92	4.63	148.27	73.62	0.14	0.03	1074914	14492
ld	87	317.38	7.94	38.48	39.97	0.09	0.21	17640896	45797
acctcms	17	294.75	6.49	14.15	45.41	0.38	0.46	2525427	5515
c2	112	289.69	9.13	34.61	31.72	0.08	0.26	3667050	9681
sh	1834	276.98	26.77	20444.24	10.35	0.01	0.00	3496613	71979
ed	524	253.13	14.46	2029.89	17.50	0.03	0.01	18058108	56039
acctprc1	3	231.28	6.67	19.45	34.67	2.22	0.34	2577344	2926
du	145	219.35	19.91	39.08	11.02	0.14	0.51	716389	23695
diff	49	175.53	6.04	25.78	29.05	0.12	0.23	3740887	11351
get	151	152.96	4.28	25.23	35.74	0.03	0.17	3634042	24917
adb	22	148.10	4.07	202.35	36.37	0.19	0.02	2313718	9813
tbl	24	143.43	2.44	210.65	58.71	0.10	0.01	1536210	3433
dd	9	139.24	10.15	51.05	13.72	1.13	0.20	26006848	294
as2	155	129.33	9.82	42.25	13.17	0.06	0.23	10500835	30165
sed	597	115.46	4.19	36.23	27.57	0.01	0.12	783825	24497
ps	51	109.69	5.92	41.55	18.54	0.12	0.14	2278056	8310
make	89	102.94	2.87	203.32	35.81	0.03	0.01	1018461	8664
delta	25	90.23	2.27	17.80	39.70	0.09	0.13	2909269	9321
cpp	172	89.37	2.69	11.32	33.19	0.02	0.24	3519054	12155
fsck	16	86.94	1.30	10.57	66.85	0.08	0.12	27671849	2927
find	52	86.64	5.05	63.87	17.15	0.10	0.08	565125	11161
ls	706	82.47	5.78	62.85	14.26	0.01	0.09	1811882	29659
xck	2	79.44	10.49	47.89	7.57	5.25	0.22	198016	21995
awk	22	78.83	1.37	5.24	57.72	0.06	0.26	355466	3769
uucico	60	75.55	1.42	632.50	53.27	0.02	0.00	398693	6377
acctcom	9	75.21	2.81	11.49	26.75	0.31	0.24	1283776	3771
echo	2814	66.10	7.08	91.80	9.33	0.00	0.08	168651	24253
ged	3	57.27	0.82	7.51	70.16	0.27	0.11	51832	426
dc	284	56.92	2.42	9.43	23.48	0.01	0.26	15283	20329
450	7	48.03	6.80	84.45	7.06	0.97	0.08	279451	1700
cat	749	45.49	5.69	478.54	8.00	0.01	0.01	8959500	27903
ntd	6	41.52	1.55	7.55	26.87	0.26	0.20	59888	478
mail	202	39.95	2.05	532.98	19.53	0.01	0.00	427217	14377
acctprc2	3	38.95	1.43	19.45	27.24	0.48	0.07	587336	87
sort	94	38.72	1.09	9.73	35.41	0.01	0.11	375876	4433
pr	104	34.89	2.47	214.50	14.10	0.02	0.01	1060989	6572
haspmain	7	33.20	5.28	1244.54	6.29	0.75	0.00	63064	36635
ex	17	31.69	0.62	41.04	50.97	0.04	0.02	514624	3593
grep	213	28.73	2.98	21.01	9.64	0.01	0.14	2100229	14297

Jun 8 04:07 1979 MONTHLY TOTAL COMMAND SUMMARY Page 1

COMMAND NAME	NUMBER CMDS	TOTAL KCOREMIN	TOTAL CPU-MIN	TOTAL REAL-MIN	MEAN SIZE-K	MEAN CPU-MIN	HOG FACTOR	CHARS TRNSFD	BLOCKS READ
TOTALS	553286	297698.78	10916.09	742924.94	27.27	0.02	0.01	820472546	26253312
nroff	1687	44681.55	995.92	5737.25	44.86	0.59	0.17	613403153	1089180
troff	1351	25692.15	583.69	4356.05	44.02	0.43	0.13	413163589	646243
spellpro	6466	17298.41	294.16	1893.79	58.81	0.05	0.16	334572640	853901
m2edit	654	13526.69	164.62	4238.58	82.17	0.25	0.04	54940426	427924
xnroff	397	10408.44	203.72	1496.32	51.09	0.51	0.14	215221419	301967
sort	7983	9292.34	226.01	2298.05	41.11	0.03	0.10	80108304	355963
c1	6139	8949.86	236.45	861.09	37.85	0.04	0.27	79897995	489661
ld	3244	8852.96	223.19	1128.09	39.67	0.07	0.20	493701995	1278119
sed	53134	8126.71	313.85	2241.78	25.89	0.01	0.14	23035033	1692990
m2find	2982	7984.45	140.18	1698.25	56.96	0.05	0.08	111330040	449604
c0	6586	7866.42	185.16	725.47	42.49	0.03	0.26	72595655	389426
ed	20083	7822.78	425.90	41898.18	18.37	0.02	0.01	483425634	1541326
tbl	660	7766.69	113.95	2458.55	68.16	0.17	0.05	50760094	83887
sh	40476	7499.67	635.00	383786.53	11.81	0.02	0.00	70525236	1421194
du	1941	6730.54	553.04	1128.44	12.17	0.28	0.49	20848359	628324
a.out	1483	5658.46	126.87	1868.87	44.60	0.09	0.07	16158675	80260
egrep	4801	5573.51	139.86	460.25	39.85	0.03	0.30	6823696	237298
lint1	793	5325.66	71.23	425.67	74.76	0.09	0.17	9599001	131592
cat	21170	4657.53	236.59	4354.24	19.69	0.01	0.05	239180412	1023965
acctprc1	42	3837.84	110.88	291.34	34.61	2.64	0.38	43954136	61123
c2	4067	3807.25	144.86	477.28	26.28	0.04	0.30	57519376	213521
grep	21212	3204.86	300.44	2727.87	10.67	0.01	0.11	139340583	899415
cpp	7469	3060.72	94.12	647.79	32.52	0.01	0.15	91471956	459882
getty	35556	2948.71	853.53	101107.45	3.45	0.02	0.01	34704751	263866
m2editD	83	2707.27	28.79	361.84	94.02	0.35	0.08	2852202	33949
as2	6454	2698.74	218.96	910.59	12.33	0.03	0.24	213336016	705690
make	1858	2449.10	64.69	4388.86	37.86	0.03	0.01	24116259	175544
ps	1034	2384.14	128.29	1207.87	18.58	0.12	0.11	54873792	204172
acctcms	294	2288.36	51.99	116.06	44.01	0.18	0.45	36124940	80523
uucico	815	2226.75	40.42	11729.01	55.08	0.05	0.00	11086105	162558
ls	18876	2170.01	152.76	1538.09	14.20	0.01	0.10	32418106	691028
find	1705	2114.18	114.35	920.75	18.49	0.07	0.12	94631199	338600
ged	72	2026.43	28.54	317.21	71.01	0.40	0.09	1648636	10374
echo	84710	2018.23	190.14	1138.49	10.61	0.00	0.17	2926992	649200
cpio	127	1956.60	77.03	391.45	25.40	0.61	0.20	190822346	296302
maze	8	1620.42	44.80	128.25	36.17	5.60	0.35	120399	212
mail	4735	1474.38	76.92	14262.62	19.17	0.02	0.01	25719618	463748
get	1085	1358.03	37.59	234.97	36.13	0.03	0.16	31540008	178623
acctcom	165	1253.99	47.06	339.34	26.64	0.29	0.14	57405662	68949
yacc	58	1187.17	15.36	36.90	77.31	0.26	0.42	4096070	12093
col	638	1064.40	49.01	2199.00	21.72	0.08	0.02	23835395	16903
line	27184	1036.03	93.14	1941.33	11.12	0.00	0.05	925447	296142
nroff1.2	29	909.83	17.71	56.97	51.38	0.61	0.31	11459920	18802
delta	264	904.54	23.07	254.06	39.21	0.09	0.09	24219141	87164
td	175	886.19	25.74	159.73	34.43	0.15	0.16	1990177	15792
ar	1434	872.65	61.87	309.07	14.11	0.04	0.20	189858731	428871
m2findD	144	864.29	12.54	344.13	68.94	0.09	0.04	1184947	28576
rm	15319	857.97	85.65	754.20	10.02	0.01	0.11	453479	433903
acctdusg	1	819.77	39.30	170.10	20.86	39.30	0.23	1812480	39744
f77pass1	155	779.13	7.97	29.09	97.70	0.05	0.27	990027	34702
diff	786	767.31	32.77	260.27	23.41	0.04	0.13	22940094	97214

Jun 8 04:07 1979 LAST LOGIN Page 1

00-00-00	dii	00-00-00	rudd	79-06-08	adm
00-00-00	absadm	00-00-00	s10	79-06-08	agp
00-00-00	absafr	00-00-00	s2	79-06-08	als
00-00-00	abscas	00-00-00	s4	79-06-08	arf
00-00-00	absjcw	00-00-00	s5	79-06-08	cath
00-00-00	abspvg	00-00-00	s6	79-06-08	dal
00-00-00	abstbm	00-00-00	s8	79-06-08	dan
00-00-00	adm94	00-00-00	s9	79-06-08	denise
00-00-00	apb	00-00-00	scbsa	79-06-08	dp
00-00-00	archive	00-00-00	sjm	79-06-08	ds52
00-00-00	asc	00-00-00	srb	79-06-08	ech
00-00-00	badt	00-00-00	sys	79-06-08	ecp
00-00-00	btb	00-00-00	tgp	79-06-08	eric
00-00-00	bvl	00-00-00	tld	79-06-08	fld
00-00-00	bwk	00-00-00	ussc	79-06-08	fsrep1
00-00-00	chicken	00-00-00	uucpa	79-06-08	games
00-00-00	class	00-00-00	uvac	79-06-08	graf
00-00-00	cleary	00-00-00	vav	79-06-08	herb
00-00-00	cs	00-00-00	wdr	79-06-08	hoot
00-00-00	dbs	00-00-00	willa	79-06-08	hsm
00-00-00	deby	00-00-00	zooma	79-06-08	jac
00-00-00	dec	79-06-04	dws	79-06-08	jcw
00-00-00	demo	79-06-04	ewb	79-06-08	jel
00-00-00	dlt	79-06-04	kas	79-06-08	jfn
00-00-00	dmr	79-06-04	satz	79-06-08	kof
00-00-00	docs	79-06-04	uucp	79-06-08	krb
00-00-00	dug	79-06-05	bcm	79-06-08	leap
00-00-00	ellie	79-06-05	lprem	79-06-08	llc
00-00-00	fsrep2	79-06-05	s7	79-06-08	m2
00-00-00	gas	79-06-05	sccs	79-06-08	m2class
00-00-00	graphics	79-06-06	conv	79-06-08	mfp
00-00-00	hjg	79-06-06	dck	79-06-08	mhb
00-00-00	hlb	79-06-06	dmt	79-06-08	mjr
00-00-00	inst	79-06-06	emp	79-06-08	msb
00-00-00	jfm	79-06-06	pah	79-06-08	nuucp
00-00-00	jrh	79-06-06	sync	79-06-08	paul
00-00-00	ken	79-06-06	tad	79-06-08	pdw
00-00-00	lco	79-06-07	ams	79-06-08	pris
00-00-00	learn	79-06-07	bin	79-06-08	pwbcs
00-00-00	lppdw	79-06-07	dgd	79-06-08	pwbst
00-00-00	lrbb	79-06-07	haight	79-06-08	rbj
00-00-00	maj	79-06-07	hasp	79-06-08	reg
00-00-00	mar	79-06-07	jgw	79-06-08	rem
00-00-00	mash	79-06-07	leb	79-06-08	rje
00-00-00	meq	79-06-07	ljk	79-06-08	rnt
00-00-00	mifi	79-06-07	mep	79-06-08	root
00-00-00	mlc	79-06-07	nhg	79-06-08	rrr
00-00-00	mmr	79-06-07	nws	79-06-08	s1
00-00-00	mpf	79-06-07	qtroff	79-06-08	s3
00-00-00	plan	79-06-07	tbm	79-06-08	star
00-00-00	plum	79-06-07	train	79-06-08	stock
00-00-00	pvg	79-06-07	whr	79-06-08	systst
00-00-00	rakesh	79-06-07	wwc	79-06-08	teach
00-00-00	rfg	79-06-08	?	79-06-08	text
00-00-00	rlc	79-06-08	abs	79-06-08	trc
00-00-00	rrc	79-06-08	absjrk	79-06-08	vjf
				79-06-08	whm

The UNIX System Activity Package

Tsyh-Wen Pao

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This memo describes the design and implementation of the UNIX† System Activity Package. This package reports UNIX system-wide statistics including CPU utilization, disk and tape I/O activities, terminal device activity, buffer usage, system calls, process switching and swapping, file-access activity, queue activity, and message and semaphore activities.

It provides four commands to generate various types of reports: *sar*, *sag*, *sadp* and *timex* commands. Procedures for automatically generating daily reports are also included.

1. INTRODUCTION

The System Activity Package reports UNIX system-wide measurements including CPU utilization, terminal device activity, disk and tape I/O activities, buffer usage, system calls, system switching and swapping, file-access activity, queue activity, and message and semaphore activities. There are five functions:

— *sar* command: allows a user to generate system activity reports in real time and to save system activities in a file for later usage.

— *sag* command: displays system activity in a graphical form.

— *sadp* command: samples disk activity once every second during a specified time interval and reports disk usage and seek distance in either tabular or histogram form.

— *timex* command: a modified *time*(1) command, which times a command and also reports concurrent system activity.

— system activity daily reports: procedures are provided for sampling and saving system activities in a data file periodically and for generating the daily report from the data file.

The system activity information reported by this package is derived from a set of system counters located in the operation system kernel. These system counters are described in Section 2. Section 3 describes the commands provided by this package. Section 4 gives the procedure for generating daily reports. A description for each of the files used by the system activity package can be found in Attachment 1.

2. SYSTEM ACTIVITY COUNTERS

The UNIX operating system manages a number of counters that record various activities and provide the basis for the system activity reporting system. The data structure for most of these counters is defined in the **sysinfo** structure (see Attachment 2) in **/usr/include/sys/sysinfo.h**. The system table overflow counters are kept in the **_syserr** structure. The device activity counters are extracted from the device status tables. In this version, the I/O activity of the following devices is recorded: RP06, RM05, RS04, RF11, RK05, RP03, RL02, TM03 and TM11.

---

† UNIX is a trademark of Bell Laboratories.

In the following paragraphs, the system activity counters that are sampled by the system activity package are described.

— **cpu** time counters: There are four time counters that may be incremented at each clock interrupt 60 times per second. Exactly one of the **cpu[ ]** counters is incremented on each interrupt, according to the mode the CPU is in at the interrupt; idle, user, kernal, and wait for I/O completion.

— **Lread** and **lwrite** count logical reads and logical writes, that is, read and write requests issued by the system to block devices.

— **Bread** and **bwrite** count blocks transferred between the system buffers and the block devices. These actual I/Os are triggered by logical I/Os that cannot be satisfied by the current contents of the buffers. The ratio of block I/O to logical I/O is a common measure of the effectiveness of the system buffering.

— **Phread** and **phwrite** count read and write requests issued by the system to raw devices.

— The **swapin** and **swapout** counters are incremented for each system request initiating a transfer from or to the swap device. More than one request is usually involved in bringing a process into memory, or out, because text and data are handled separately. Commonly used programs are kept on the swap device and are swapped in rather than loaded from the file system. The **swapin** counter reflects these initial loading operations as well as resumptions of activity, while the **swapout** counter reveals the level of actual "swapping." The amount of data transferred between the swap device and memory are measured in blocks and counted by **bswapin** and **bswapout**.

— Counters **syscall** and **pswitch** are related to the management of multiprogramming. **Syscall** is incremented every time a system call is invoked. The numbers of invocations of system calls: *read*, *write*, *fork* and *exec*, are kept in counters **sysread**, **syswrite**, **sysfork** and **sysexec**.

Pswitch counts the times the switcher was invoked, which occurs when:

a. a system call resulted in a road block,
b. an interrupt occurred resulting in awakening a higher priority process, or
c. 1 second clock interrupt.

— Counters **iget**, **namei**, and **dirblk** apply to file-access operations. **Iget** and **namei**, in particular, are the names of UNIX operating system routines; the counters record the number of times that the respective routines are called. **Namei** is the routine that performs file system path searches. It searches the various directory files to get the associated i-number of a file corresponding to a special path. **Iget** is a routine called to locate the inode entry of a file (i-number). It first searches the in-core inode table. If the inode entry is not in the table, routine **iget** will get the inode from the file system where the file resides and make an entry in the in-core inode table for the file. **Iget** returns a pointer to this entry. **Namei** calls **iget**, but other file access routines also call **iget**. Therefore, counter **iget** is always greater than counter **namei**.

Counter **dirblk** records the number of directory block reads issued by the system. It is noted that the directory blocks read divided by the number of **namei** calls estimates the average path length of files.

— **Runque**, **runocc**, **swpque** and **swpocc** record queue activities. They are implemented in the **clock.c** routine. At every one second interval, the clock routine examines the process table to see whether any processes are in core and in ready state. If so, the counter **runocc** is incremented and the number of such processes are added to counter **runque**. While examining the process table, the clock routine also checks whether any processes in the swap device are in ready state. The counter **swpocc** is incremented if the swap queue is occupied and the number of processes in swap queue is added to counter **swpque**.

— **Readch** and **writech** record the total number of bytes (characters) transferred by the *read* and *write* system calls respectively.

— There are six counters monitoring terminal device activities. **Rcvint**, **xmtint** and **mdmint** are counters measuring hardware interrupt occurrences for receiver, transmitter and modem individually. **Rawch**, **canch** and **outch** count number of characters in the raw queue, canonical queue and output queue. Characters generated by devices operating in the *cooked* mode, such as terminals, are counted in both **rawch** and (as edited) in **canch**, but characters from raw devices, such as communication processors, are counted only in **rawch**.

— Counters **msg** and **sema** record message sending and receiving activities and semaphore operations, respectively (refer to manual entries *msg*(2) and *sema*(2)).

— As to the I/O activity for a disk or tape device, four counters are kept for each disk or tape drive in the device status table. Counter **io_ops** is incremented when an I/O operation has occurred on the device. It includes block I/O, swap I/O and physical I/O. **Io_bcnt** counts the amount of data transferred between the device and memory in blocks. **Io_act** and **io_resp** measure the active time and response time of a device in time ticks. The device active time includes the device seeking, rotating and data transferring times, while the response time of an I/O operation is from the time the I/O request is queued to the device, to the time when the I/O completes.

— Counters **inodeovf**, **fileovf**, **textovf** and **procovf** are extracted from **_syserr** structure. When an overflow occurs in any of the inode, file, text and process tables, the corresponding overflow counter is incremented.

3. SYSTEM ACTIVITY COMMANDS

The System Activity Package provides three commands for generating various system activity reports and one command for profiling disk activities. These tools facilitate observation of system activity:

— during a controlled stand alone test of a large system,

— during an uncontrolled run of a program to observe the operating environment, and

— during normal production operation.

Commands *sar* and *sag* permit the user to specify a sampling interval and number of intervals for examining system activity, and then to display the observed level of activity in tabular or graphical form. The *timex* command reports the amount of system activity that occurred during the precise period of execution of a timed command. The *sadp* command allows the user to establish a sampling period during which access location and seek distance on specified disks are recorded and later displayed as a tabular summary or as a histogram.

— *Sar* command:
It can be used in two ways:

- When the frequency arguments *t* and *n* are specified, it invokes the data collection program *sadc* to sample the system activity counters in the operating system every *t* seconds for *n* intervals and generates system activity reports in real time. Generally it is desirable to include the option to save the sampled data in a file for later examination.

  The format of the data file is shown in *sar*(8). In addition to the system counters, a time stamp is also included. It gives the time at which the sample was taken.

- If no frequency arguments are supplied, it generates system activity reports for a specified time interval from an existing data file that was created by *sar* at an earlier time.

A convenient usage is to run *sar* as a background process, saving its samples in a temporary file, but sending its standard output to /dev/null. Then an experiment is conducted, after

which the system activity is extracted from the temporary file. The *sar*(1) manual entry describes the usage and lists various types of reports. Attachment 3 gives formulae for deriving each reported item.

— *Sag* command:
Sag displays system activity data graphically. It relies on the data file produced by a prior run of *sar*, after which any column of data, or the combination of columns of data of the *sar* report can be plotted. A fairly simple but powerful command syntax allows the specification of cross plots or time plots. Data items are selected using the *sar* column header names. The *sag*(1G) manual entry describes its options and usage.

The system activity graphical program invokes *graph* and *tplot* commands to have the graphical output displayed on any of the terminal types supported by *tplot*.

— *Timex* command:
The *timex* command is an extension of the *time*(1) command. In addition to giving the time information, it also prints a system activity report derived from the system counters.

The manual entry *timex*(1) explains its usage. It should be emphasized that the **user** and **sys** times reported in the second and third lines are for the measured process itself including all its children, while the remaining data (including the **cpu user %** and **cpu sys %**) are for the entire system.

While the normal use of *timex* will probably be to measure a single command, multiple commands can also be timed; either by combining them in an executable file and timing it, or more concisely, by typing:

```
timex sh -c "cmd1; cmd2; ... ;"
```

This establishes the necessary parent-child relationships to correctly extract the user and system times consumed by *cmd1*, *cmd2*, ... (and the shell).

— *Sadp* command:
Sadp is a user level program that can be invoked independently by any user. It requires no storage or extra code in the operating system and allows the user to specify which disks are to be monitored. The program is reawakened every second, reads system tables from **/dev/kmem**, and extracts the required information. Because of the one second sampling, only a small fraction of disk requests are observed, however, comparative studies have shown that the statistical determination of disk locality is adequate when sufficient samples are collected.

In the operating system, there is an *iobuf* for each disk drive. It contains two pointers which are head and tail of the I/O active queue for the device. The actual requests in the queue may be found in three buffer header pools: system buffer headers for block I/O reguests, physical buffer headers for physical I/O reguests and swap buffer headers for swap I/O. Each buffer header has a forward pointer which points to the next request in the I/O active queue and a backward pointer which points to the previous request.

Sadp snapshots the *iobuf* of the monitored device and the three buffer header pools once every second during the monitoring period. It then traces the requests in the I/O queue and records the disk access location and seek distance in buckets of 8 cylinder increments. At the end of monitoring period, it prints out the sampled data. The output of *sadp* can be used to balance load among disk drives and to rearrange the layout of a particular disk pack. The usage of this command is described in manual entry *sadp*(1).

4. DAILY REPORT GENERATION

The previous section described the commands available to users to initiate activity observations. It is probably desirable for each installation to routinely monitor and record system activity in a

standard way for historical analysis. This section describes the steps that a system administrator may follow to automatically produce a standard daily report of system activity.

— Facilities:

- *sadc* — the executable module of **sadc.c** (see Attachment 1) which reads system counters from **/dev/kmem** and records them to a file. In addition to the file argument, two frequency arguments are usually specified to indicate the sampling interval and number of samples to be taken. In case no frequency arguments are given, it writes a dummy record in the file to indicate a system restart.
- *sa1* — the shell procedure that invokes *sadc* to write system counters in the daily data file **/usr/adm/sa***dd* where *dd* represents the day of the month. It may be invoked with sampling interval and iterations as arguments.
- *sa2* — the shell procedure that invokes the *sar* command to generate daily report **/usr/adm/sa/sar***dd* from the daily data file **/usr/adm/sa/sa***dd*. It also removes daily data files and report files when they are over 7 days old. The starting and ending times and all report options of *sar* are applicable to *sa2*.

— Suggested operational setup:
It is suggested that the *cron*(1M) control the normal data collection and report generation operations. For example, the sample entries in **/usr/lib/crontab**:

```
0 * * * 0,6 su sys -c "/usr/lib/sa/sa1"
0 18-7 * * 1-5 su sys -c "/usr/lib/sa/sa1"
0 8-17 * * 1-5 su sys -c "/usr/lib/sa/sa1 1200 3"
```

would cause the data collection program *sadc* to be invoked every hour on the hour. Moreover, depending on the arguments presented, it writes data to the data file once or 3 times at every 20 minutes. Therefore, under the control of *cron*(1M), the data file is written every 20 minutes between 8:00 and 18:00 on weekdays and hourly at other times.

Note that data samples are taken more frequently during prime time on weekdays to make them available for a finer and more detailed graphical display. It is suggested that *sa1* be invoked hourly rather than invoking it once every day, this ensures that no matter when the system crashes, the data will be collected within an hour after the system is restarted.

Because system activity counters restart from zero when the system is restarted, a special record is written on the data file to reflect this situation. This process is accomplished by invoking within **/etc/rc** when going to multi-user state:

```
su adm -c "/usr/lib/sa/sadc /usr/adm/sa/sa`date +%d`"
```

Cron(1M) also controls the invocation of *sar* to generate the daily report via shell procedure *sa2*. One may choose the time period at which the daily report is to cover and which groups of system activity are to be reported. For instance, if:

```
0 20 * * 1-5 su sys -c "/usr/lib/sa/sa2 -s 8:00 -e 18:00 -i 3600 -uybd"
```

is an entry in **/usr/lib/crontab**, *cron* will execute the *sar* command to generate daily reports from the daily data file at 20:00 on weekdays. The daily report reports the CPU utilization, terminal device activity, buffer usage and device activity every hour from 8:00 to 18:00.

In case of a shortage of the disk space or for any other reason, these data files and report files can be removed by the super-user. The manual entry *sar*(8) describes the daily report generation procedure.

5. ACKNOWLEDGEMENTS

L. A. Wehr is responsible for the set of system activity counters incorporated in the UNIX Time-Sharing System. The author wishes to acknowledge his discussions and help in providing a test environment during the development of the *sar* command. The output format of the *sadp* command is adopted from *iostat* of CB/UNIX. Thanks are due to T. Cook and D. DeJager for their cooperation in making the verification of the result generated by *sadp* possible. Finally, the author gratefully acknowledges D. A. DeGraaf's contribution to this package and appreciates A. Petruccelli's efforts in making this memo in its printable format.

ATTACHMENT 1

The source files and shell programs of the system activity package are in directory **/usr/src/cmd/sa**.

sa.h	the system activity header file which defines the structure of data file and device information for measured devices. It is included in **sadc.c**, **sar.c** and **timex.c**.
sadc.c	the data collection program that accesses **/dev/kmem** to read the system activity counters and writes data either on standard output or on a binary data file. It is invoked by the *sar* command generating a real time report. It is also invoked indirectly by entries in **/usr/lib/crontab** to collect system activity data.
sar.c	the report generation program that invokes *sadc* to examine system activity data and generate reports in real time, and save the data to a file for later usage. It may also generate system activity reports from an existing data file. It is invoked indirectly by *cron* to generate daily reports.
saghdr.h	the header file for **saga.c** and **sagb.c**. It contains data structures and variables used by **saga.c** and **sagb.c**.
saga.c & sagb.c	the graph generation program that first invokes *sar* to format the data of a data file in a tabular form, and then displays the *sar* data in graphical form.
sa1.sh	the shell procedure that invokes *sadc* to write data file records. It is activated by entries in **/usr/lib/crontab**.
sa2.sh	the shell procedure that invokes *sar* to generate the report. It also removes the daily data files and daily report files when they are a week old. It is activated by an entry in **/usr/lib/crontab** on weekdays.
timex.c	the program that times a command and generates a system activity report.
sadp.c	the program that samples and reports disk activities.

ATTACHMENT 2

```
struct sysinfo {
        time_t          cpu[4];
#define CPU_IDLE        0
#define CPU_USER        1
#define CPU_KERNAL      2
#define CPU_WAIT        3
        time_t          wait[3];
#define W_IO            0
#define W_SWAP          1
#define W_PIO           2
        long            bread;
        long            bwrite;
        long            lread;
        long            lwrite;
        long            phread;
        long            phwrite;
        long            swapin;
        long            swapout;
        long            bswapin;
        long            bswapout;
        long            pswitch;
        long            syscall;
        long            sysread;
        long            syswrite;
        long            sysfork;
        long            sysexec;
        long            runque;
        long            runocc;
        long            swpque;
        long            swpocc;
        long            iget;
        long            namei;
        long            dirblk;
        long            readch;
        long            writech;
        long            rcvint;
        long            xmtint;
        long            mdmint;
        long            rawch;
        long            canch;
        long            outch;
        long            msg;
        long            sema;
};
```

ATTACHMENT 3

The derivation of the reported items of a report is given in this attachment. Each item discussed below is the data difference sampled at two distinct times *t2* and *t1*.

— CPU utilization:

%-of-cpu-x = cpu-x / (cpu-idle + cpu-user + cpu-kernel + cpu-wait) * 100

where cpu-x is cpu-idle, cpu-user, cpu-kernel (cpu-sys) or cpu-wait.

— Cached hit ratio:

%-of-cached-I/O = (logical-I/O − block-I/O) / logical-I/O * 100

where cached I/O is cached read or cached write.

— disk or tape I/O activity:

%-of-busy = I/O-active / (t2 − t1) * 100;
avg-queue-length = I/O-resp / I/O-active;
avg-wait = (I/O-resp − I/O-active) / I/O-ops;
avg-service-time = I/O-active / I/O-ops.

— queue activity:

avg-x-queue-length = x-queue / x-queue-occupied-time;
%-of-x-queue-occupied-time = x-queue-occupied-time / (t2 − t1);

where x-queue is run queue or swap queue.

— The rest of system activity:

avg-rate-of-x = x / (t2 − t1)

where x is swap in/out, blks swapped in/out, terminal device activities, read/write characters, block read/write, logical read/write, process switch, system calls, read/write, fork/exec, iget, namei, directory blocks read, disk/tape I/O activities, message or semaphore activities.

January 1981

A Stand-Alone Input/Output Library

S. R. Eisen

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

1.1 Motivation

Most stand-alone programs that are supported under UNIX† conform to no input-output standard. They implement their own I/O routines and their own nomenclature for accessing data stored on I/O devices. This library was written with the objective of creating a set of functions that would be used to simulate standard C library functions [1] for a program that is loaded stand-alone into a Digital Equipment Corporation 11-family computer.

1.2 Environment

1.2.1 Compilation and Execution. Normally, a stand-alone program is written in C, using standard library functions found in Sections 2 and 3 of [1]. The program is compiled and the object file is link-edited with the stand-alone library *instead* of the standard UNIX C library. The resulting single object file is loaded by using either the command interpreter that is described in Section 6.2.1 (denoted below by {6.2.1}), or any other standard UNIX bootstrap program.

1.2.2 System Functions. All required services that are usually performed by the operating system, such as input/output, are taken care of by the functions loaded from the stand-alone library. Thus I/O drivers are included in stand-alone executables without any additional work on the part of the user.

Functions such as `fork`, `pipe`, and `exec`, that would simulate system calls that make no sense outside of an operating system environment are *excluded* from the stand-alone library, even to the extent of signifying an error condition. The complete list of excluded functions may be found in {4.3}. Of the routines that were *substituted* for UNIX system calls, all take the same arguments and return the same values as their UNIX counterparts, except as noted in {4.1-4.2}. These functions set the external variable `errno` when an error occurs, so that the C library routine `perror` may be used by stand-alone programs.

The user may call any global functions in the library, including those that would normally be found in an operating system proper, but would not be available to the user in an operating system environment. All such routines, however, have been "disguised" by prefacing their names with the underscore character.

1.2.3 User Interface. UNIX-like file names need not be used, although their use is encouraged. All user functions that require file names, such as `MKNOD` {2.1.2}, `mount` {2.1.3}, and `open` {2.1.4} first pass their file-name arguments through a filter that converts them to a standard form: each element of the path name is separated by a single slash, with a leading slash used only if the file name is non-null.

From the point of view of the user-level program, the environment that is created by the stand-alone library is close enough to a UNIX environment that *a large class of UNIX programs may be compiled for stand-alone execution with little or no revision.* Another class of programs that includes boot programs and other programs that need to be relocated can also be written using the stand-alone I/O library. Specific instructions for compiling and executing programs using the library may be found in {6}.

---

† UNIX is a trademark of Bell Laboratories.

2. I/O PHILOSOPHY

The stand-alone I/O library was designed to provide an environment that is as close to UNIX as possible, while maintaining the generality necessary for the composition of bootstrap programs, disk formatters, and the like. Disk I/O drivers have the capability of handling UNIX file systems, but retain the generality necessary to manipulate disks with other data on them. Because UNIX accesses I/O devices through the file system, and there is no guarantee that a file system (UNIX or otherwise) exists, access to I/O devices must be handled in a special way.

2.1 Block I/O Data Structures

2.1.1 The Configuration Table. All I/O routines operate without interrupt processing; also, the stand-alone implementation of file descriptors differs from the UNIX implementation. The open, close, and strategy (read and write) routines for devices therefore do not strictly resemble the corresponding UNIX routines. The method used to access these routines, however, is very similar; it employs a configuration table that has the form:

```
struct devsw {
        int     (*dv_strategy) ();
        int     (*dv_open) ();
        int     (*dv_close) ();
};
```

The position of a certain device within the `devsw` table is the *device number* for that type of device. The notion of a device number is analogous to the notion of a major number for a UNIX device.

2.1.2 The Device Table. Each family of devices is associated with UNIX-type names by use of a second structure:

```
struct dtab {
        char            *dt_name;
        struct devsw    *dt_devp;
        int             dt_unit;
        daddr_t         dt_boff;
};
```

The `dtab` structure associates a device name with a pointer to the `devsw` structure for that type of device, the unit number of the physical device, and the block offset within the unit at which the logical device should start. The name, in fact, can be any string; by convention, however, a UNIX-type file name, such as */dev/rk1* or */dev/mt4*, is used.

Entries in this table are created by using the MKNOD function. Note that although the function of the MKNOD routine is similar to the that of the UNIX `mknod` routine, the arguments passed to each routine are not at all alike. MKNOD is called using the following synopsis:

```
MKNOD (name, devno, unit, boff)
char *name;
int devno, unit;
daddr_t boff;
```

MKNOD associates `name` with the logical device beginning `boff` blocks into the given unit of the device whose device number is `devno`. The value -1 is returned if an illegal argument is passed, the `dtab` table is full, or the given name already exists in the table.

2.1.3 The Mount Table. For mounted file systems, there is yet another structure:

```
struct mtab {
        char            *mt_name;
        struct dtab     *mt_dp;
};
```

The mtab structure associates a name with a pointer to the dtab structure for a device on which a UNIX file system resides. References to the name will refer to the root file on that device. Entries in this table are created by using the mount function. The following synopsis applies:

```
mount (devname, mntname)
char *devname, *mntname;
```

Mount announces that a file system has been mounted on devname, and that its mounted name will henceforth be mntname. Devname must be a valid entry in the dtab table, and mntname must *not* exist in the mtab table. If either of these conditions is not met, or if there are no more empty slots in the table, mount returns the value −1.

A mount table entry may be deleted by the umount function, whose synopsis is the same as the corresponding UNIX routine.

2.1.4 The I/O Block. Each open file is associated with a numerical file descriptor. At the start of program execution, the file descriptors numbered 0, 1 and 2 are each open for reading and writing to the system console, and all other file descriptors are closed (not assigned). File descriptors greater than 2 are available to be assigned to either block devices or UNIX files that reside on mounted file systems by using the open function described below.

Each block file descriptor is associated with a structure of the following form:

```
struct iob {
        char            i_flgs;
        struct inode    i_ino;
        time_t          i_atime;
        time_t          i_mtime;
        time_t          i_ctime;
        struct dtab     *i_dp;
        off_t           i_offset;
        daddr_t         i_bn;
        char            *i_ma;
        int             i_cc;
        char            i_buf[512];
};
```

The I/O block contains a data buffer and a block number counter for the device whose dtab structure is pointed to by the I/O block. For open UNIX files, the offset within the file and a copy of the inode are included in the I/O block. For open block devices, the inode structure is only partially filled in.

A file descriptor is allocated and an entry is created in the I/O block by the open function. The synopsis of the stand-alone open function is identical with that of its UNIX counterpart.

Open searches the dtab table for the given string, and if it is not found, the mtab table is searched for the longest path name starting at the beginning of the given string. For example, if open is passed the argument /ab/cd/ef/gh, it will first look for the argument itself in both the dtab and mtab tables, then search for /ab/cd/ef in the mtab table, then /ab/cd, and so on.

If the string is found in the dtab table, then the named device will be opened for the appropriate operation. If the string or one of its substrings is found in the mtab table, the device pointed to by the mtab table entry is searched for the remainder of the path name. If found, the file is opened.

At present, files on mounted file systems may only be opened for reading. The reason for this has to do with memory size requirements for a writing capability, the amount of time it would take to implement this capability, and the danger of corrupting file systems unnecessarily. It is likely that the capability of writing files will be included at some time in the future.

The `creat (name, mode)` function is identical to `open (name, 1)`. The `mode` argument is ignored.

The `close` function deallocates the I/O block associated with the named file descriptor.

2.1.5 Summary. The following list contains the definitions of all of the data structures discussed in this section, as they appear in the stand-alone library source code:

```
struct devsw    _devsw[];
struct dtab     _dtab[NDEV];
struct mtab     _mtab[NMOUNT];
struct iob      _iobuf[NFILES];
```

These structures and the corresponding table sizes are all defined in the file `/usr/include/stand.h`.

2.2 Reading and Writing

The `read` and `write` functions are the most primitive I/O routines normally available to the user. The file descriptor argument may refer to either the system console or a block device.

3. I/O DEVICES AND DRIVERS

As was mentioned earlier, file descriptors 0, 1, and 2 all refer to the system console device. The console is the only character device supported. A spectrum of block devices may be defined in the device table by the MKNOD function.

3.1 The System Console Driver

The driver for the console terminal is a modified, scaled-down version of the UNIX *tty* driver. Input lines may be up to 255 characters long and there is no read-ahead (i.e., input will not be accepted until the program calls for it). The driver supports programmable options and *erase* and *kill* characters. End of file may be generated in "cooked" mode by typing CTRL-D.

The `stty` and `gtty` functions are implemented and refer to a structure identical with that which is used by UNIX. The only options that have any effect are RAW, CRMOD, XTABS, ECHO, and LCASE. Initially, the *erase* and *kill* characters are the standard UNIX # and @, respectively, and the options set are CRMOD, XTABS, and ECHO.

The `isatty` function returns true if the file descriptor argument is in the range 0 to 2.

If, while output is being printed on the console, the ASCII DEL character is typed, a subroutine call to the `_exit` function is immediately effected.

The actual input and output are performed by the functions in the following table:

System Console Driver Routines

Synopsis	Description

`_ttread (buf, n)`
`char *buf;`
`int n;`
	Reads `n` characters from the console into the area pointed to by `buf`.

`_ttwrite (buf, n)`
`char *buf;`
`int n;`
	Prints `n` characters on the console from the area pointed to by `buf`.

The external buffer `_ttstat` contains the current copy of the structure referred to by `stty` and `gtty`. Its synopsis is:

```
# include <stand.h>
struct sgttyb _ttstat;
```

3.2 Block Device Drivers

Block input and output are performed in the stand-alone library in the same manner as *physical* I/O is handled under UNIX; that is, only raw devices are supported.

A particular I/O driver routine is looked up in the `devsw` table and called by one of the following:

```
_devopen (io)      _devclose (io)     _devread (io)      _devwrite (io)
struct iob *io;    struct iob *io;    struct iob *io;    struct iob *io;
```

The external integer variable `_devcnt` contains the number of devices in the `devsw` table.

3.2.1 Disk Drivers. The stand-alone library supports the following disk devices and their equivalents:

RP04/05/06 and RM05(*gd*) RP11/RP03(*rp*) RK11/RK05(*rk*)

Disk device drivers can support file systems that do not start at the beginning of the physical unit. Such file systems are defined by using the MKNOD function {2.1.2}.

The physical I/O operation for disks causes reads and writes to always be started at the beginning of the physical block in which the offset designated in the I/O block {2.1.4} falls. Also, I/O operations that reference a disk address outside of the bounds of either a logical or physical disk will not cause an error to occur.

The synopsis of each of the disk driver functions has the form:

```
_devstrategy (io, func)
struct iob *io;
int func;
```

where *dev* may be `gd`, `rp`, or `rk`.

3.2.2 Tape Drivers. The stand-alone library supports the following magnetic tape devices and their equivalents:

TM11/TU10(*tm*) TU16(*ht*)

For both the *tm* and *ht* drivers, logical units 0 through 7 refer to four 800 bpi magnetic tape transports. For the *ht* driver only, logical units 8 through 15 refer to the corresponding 1600 bpi magnetic tape transports. In each block of eight logical units, the first four units are designated normal-rewind on close, and the other four are no-rewind on close.

`Lseek` is ineffective for tapes. Each `read` or `write` function call reads or writes the next record on the tape. The `dt_boff` entry in the device table is ignored for magnetic tape devices.

The synopses of the tape driver functions have the following forms:

```
_devopen (io)      _devclose (io)     _devstrategy (io, func)
struct iob *io;    struct iob *io;    struct iob *io;
                                      int func;
```

where *dev* may be either `ht` or `tm`.

4. NON-I/O ROUTINES

4.1 Revisions

Several of the system calls that are not required for I/O, but would, however, be useful in a stand-alone environment are included in the library. The operation of some of these functions may differ slightly from the UNIX implementations. These functions, together with the I/O functions described above, form a firm enough basis that the remainder of the C library may be used without modification.

4.1.1 Stat and Fstat. The `stat` and `fstat` functions require the use of an I/O block. In order to execute either one of these functions, the file on which it is operating must be open because the information needed is copied out of the I/O block. For `fstat`, the file is already open, but when the `stat` function is used, first the file is opened, `fstat` is called, and then the file is closed again before returning. Thus, if all I/O blocks are occupied (the maximum number of files are open), `stat` will return an error.

If the argument to `stat` or `fstat` refers to a file that resides on a mounted file system, then the inode is copied verbatim and the routines are completely compatible with the UNIX versions. If the argument refers to a device, the buffer is filled with a reasonable approximation of what may be expected.

4.1.2 Access. The `access` function also requires an open file. If the open succeeds, and either the file is a device or the mode of the file matches the specified mode argument, the value 0 is returned; otherwise, the value −1 is returned. In any case, the I/O block is freed by closing the file before returning.

4.1.3 Time. Because the real-time clock is not supported, the best that can be done is for the `time` function to return the value that was set by the last call of the `stime` function. If `stime` has not been called, `time` returns the value 0.

4.1.4 Break. The `brk` and `sbrk` functions may be used as they are in UNIX. Because memory management is not used, there is no way of detecting if the upward-expanding allocated memory has collided with the downward-expanding stack. The return is therefore always successful, even if the memory allocation request was too large.

4.1.5 Ustat. The `ustat` function takes as its first argument the offset of the device within the `dtab` table. This value is returned by `stat` and `fstat`, when given a device argument, in the `st_dev` and `st_rdev` buffer entries {4.1.1}. The stand-alone `ustat` returns the same information as the UNIX version.

4.1.6 Chdir. The global character pointer `_cdir` is set to the given string, which is prefixed to all file names not beginning with a slash. The string need not be a valid directory name, so `chdir` always returns successfully.

4.1.7 Lseek and Tell. There are no differences between the execution of these stand-alone functions and the operation of the corresponding UNIX routines.

4.1.8 Exit. The functions `exit` and `_exit` have the same meanings as they do in UNIX. The `_exit` function will attempt to return to the bootstrap program directly, and the `exit` function will call the `_cleanup` function first. The user may define his own `_cleanup` function or use the standard `_cleanup` that would be loaded from the library.

4.2 Null Functions

Several functions are included in the stand-alone library that only return zero or error values. These functions were included in the library to resolve external references in some C library functions. The functions that return a value of 0 are:

```
getgid  getegid  getuid  geteuid  nice  umask
```

The `chmod` function returns an error.

4.3 Deletions

The following is a complete list of those C library modules that have *not* been included in the stand-alone library:

```
acct.o      execvp.o    maus.o      sema.o      sync.o
alarm.o     fcntl.o     mktemp.o    setgid.o    syscall.o
cerror.o    fork.o      msg.o       setpgrp.o   system.o
chown.o     fp.o        oldmsg.o    setuid.o    tempfile.o
chroot.o    getpass.o   pause.o     signal.o    times.o
dup.o       getpid.o    pipe.o      sleep.o     ulimit.o
execl.o     getppid.o   plock.o     smclose.o   uname.o
execle.o    ioctl.o     popen.o     smfree.o    unlink.o
execv.o     kill.o      profil.o    smget.o     utime.o
execve.o    link.o      ptrace.o    smopen.o    wait.o
```

Several of these functions may, indeed, be faked rather than excluded; it is likely that the size of this list will be decreased in the future.

5. UTILITY FUNCTIONS

The functions described in this section do not have equivalent functions implemented in the C library.

5.1 User Functions

The following routines are included in the stand-alone library for the convenience of the user.

5.1.1 Getargv. The user has the option of having his stand-alone program invoked by a command interpreter program {6.2.1}, or by another standard UNIX bootstrap program. When a stand-alone program is not invoked by the command interpreter program, there can be no arguments specified on a command line and, consequently, no `argc`, `argv`, or environment are available to be passed to the program. In this case, the start-up code loads a value of 1 into `argc`, a null string into `argv[0]`, and a pointer to a null environment list into `envp`.

The `getargv` functions allows the program to pick up arguments after execution of the main routine has begun. The synopsis is:

```
getargv (cmd, argvp, ff)
char *cmd, *(*argvp[]);
int ff;
```

A prompt and the `cmd` argument are printed on the console and one line is read from the console. The *space* and *tab* characters are considered to be delimiters, and the single quote and double quote characters are properly understood. The arguments are stored in `argv`-format, with `cmd` as `argv[0]`, and the value of `argv` itself is stored into the address pointed to by `argvp`. The value of `argc` is returned.

Note that the area of memory used for the `argv` list is allocated by calling the `malloc` library function. A non-zero value for `ff` causes `getargv` to call the `free` function for `argvp` before calling `malloc`. The value of the `ff` argument would normally be zero on the first and only the first call.

If a typing error is made as the command is being entered, and the *kill* character is typed with the intention of retyping the line, there is a certain temptation to retype not only the arguments, but the command, too. Caveat.

5.1.2 Init. Before the `main` routine is called by the start-up code, the `_init` function is called. Normally, this function does some standard MKNODs and `mounts`, but the user can define his own `_init`, if he does not want the standard one to be loaded. The synopsis is:

```
_init ()
```

5.2 System Functions

The following external functions form that part of the kernel of the stand-alone "system" that were globally defined for the purpose of communication within the modules of the system. Several may be useful to the user, but most will not be, and are included here for the sake of completeness:

System Utility Functions

Synopsis	Description
`_cond (istr, ostr)` `char *istr, *ostr;`	Converts `istr` into the form described in {1.2}, prepends the string given to `chdir`, if any, and places the result in the buffer pointed to by `ostr`.

`ino_t`
`_find (path, io)`
`char *path;`
`struct iob *io;`
	Returns the inode number of the proper path name pointed to by `path` on the file system described in `io`, and fills the appropriate parts of the `io` structure.

`_openi (n, io)`
`ino_t n;`
`struct iob *io;`
	Fills the `inode` structure in `io` with a copy of the disk inode whose number is `n` on the file system described in `io`.

`_prs (str)` `char *str;`	Prints the simple character string `str` on the console immediately.

`daddr_t`
`_sbmap (io, bn)`
`struct iob *io;`
`daddr_t bn;`
	Returns the number of the physical block corresponding to the logical block `bn` of the file on the device described in `io`.

`_trap (ps)` `int ps;`	Prints the type of trap that has occurred, based on the passed value of `ps`.

6. COMPILING AND EXECUTING STAND-ALONE PROGRAMS

6.1 Compilation

Programs are normally prepared for stand-alone execution by the UNIX `scc` command. The syntax of this command is a superset of the standard `cc` command:

`scc [ +[` *lib* `] ] [` *option* `] ... [` *file* `] ...`

The *option* and *file* arguments may be anything that can legally be used with the `cc` command; it should be noted, though, that the `-p` (profiling) option, as well as any object module that contains system calls, will cause the executable not to run.

`Scc` defines the compiler constant, `STANDALONE`, so that sections of C programs may be compiled conditionally for when the executable will be run stand-alone.

The first argument to `scc` specifies an auxiliary library that defines the device configuration of the computer for which the stand-alone executable is being prepared. On the PDP-11, *lib* may be either one of the following; on the VAX-11/780, *lib* may only be `A`:

`A` RP04/05/06 disk (also, RM05 disk on the VAX) and TU16 magnetic tape, or equivalent

`B` RK11/RK05 disk, RP11/RP03 disk, and TM11/TU16 magnetic tape, or equivalent

If no *+lib* argument is specified, `+A` is assumed. If the `+` argument is specified alone, no configuration library is loaded unless the user supplies his own. A manual entry for the `scc` command may be found in [1].

The user may define his own configuration library by loading an object module that defines `_devsw` to be an array of `devsw` structures {2.1.1}, `_devcnt` to be the number of structures in the array {3.2}, and `_init` to be a function that is to be called before the `main` routine {5.1.2}. If the user only wishes to define his own `_init` and not `_devsw` and `_devcnt`, or vice versa, he may do so, but the configuration library must also be loaded in order to resolve the other external reference(s).

6.2 Execution

6.2.1 Sash. Stand-alone programs are normally loaded using a command interpreter which passes the arguments that it reads after its prompt into the loaded program's `argv` list. This command interpreter is called `sash` (for *s*tand-*a*lone *sh*ell). Its implementation is described here, and its use is described more completely in the Appendix.

`Sash` relocates itself up 64K words on a PDP-11, and 320K words on a VAX-11/780. This enables a stand-alone user program to use all of memory below it.

Normally, only programs with execution modes 407 and 410 may be executed (see *a.out*(5) in [1]). On the PDP-11, `sash` turns on memory management in order to relocate itself, and then executes the high-memory copy of itself in *user* mode. It loads the user's program into low memory, copies the argument list to the upper limit of addressability for a non-separate instruction/data space program, sets up a small program beneath the argument list that interfaces from the user's program (which runs in *kernel* mode) to `sash` and sets the *kernel* stack pointer to its initial value, which is just underneath the small interface program; `sash` then manages to begin execution of the user's program in *kernel* mode at physical location 0. The interface program enables the user's program to return (exit) back to `sash` by a simple `rts` instruction. The use of memory management normally allows the user's program about 55.6K words for *text*, *data*, and *bss* segments. If the user wishes to set up his own *bss* segment, then only *text* and *data* are limited to 55.6K words. It should be noted, however, that because memory management is enabled at the outset, the user's program must turn memory management off before changing any memory management-related registers.

To load mode 411 (separate instruction and data space) files, `sash` loads the *data* and *bss* segments at physical address 0 (set to be *kernel data*), and the *text* segment is loaded at the next 64-byte boundary (set to be *kernel text*). `Sash` then turns off memory management, and assumes that the program will restructure itself. It cannot be run without restructuring because the program break can only expand onto the *text* segment, and the stack pointer may contain an address that is in the middle of the *text* segment.

The address space of the VAX-11/780 is sufficiently large that memory management need not be used, and the user's program may be started by a simple subroutine call, and exited by a return from that call.

6.2.2 Other Bootstrap Programs. Alternatively, a stand-alone program may be loaded into memory by some other UNIX bootstrap program. If this is done, the start-up code senses that an argument list is not available, so `argc` will be set to 1, and `argv[0]` will be set to a null string before execution begins, and may be reassigned by `getargv`.

6.3 Relocatable Programs

The stand-alone I/O library may be used with programs than need to relocate themselves at some point during execution. Although this is never a simple task, it is quite a bit easier to do so on the VAX-11/780 than the PDP-11, and somewhat easier on the PDP-11 if memory management need not be used. The user who is considering writing a relocatable program is referred to the source code of the machine-dependent (assembler language) part of the `sash` program {Appendix} for hints.

On the VAX-11/780, the `-T` option may be given to the `ld` program to do the relocation. On the PDP-11, no special processing by `ld` is necessary.

7. OVERHEAD AND PERFORMANCE

On both the PDP-11 and VAX-11/780, a null program will compile to produce an executable object module that has a *text* segment that is slightly larger than 6K bytes, and *data* and *bss* segments that add up to about 8K bytes. This is a good rule-of-thumb calculation for the minimum size of a program that is compiled with the stand-alone library.

Because stand-alone programs run (by definition) without competing against other processes for CPU time, and are never swapped out of memory, a stand-alone program's execution is faster than that of the same program running under UNIX. However, if that program does some I/O operations, it will not benefit from some of the short-cut operations that are implemented in UNIX, such as disk read-ahead, and will therefore actually run more slowly stand-alone than under UNIX.

Acknowledgements

The stand-alone I/O library was originally based on a library written by Charles Haley whom I would like to thank for his comments and suggestions during the course of my work. I would also like to thank Larry Wehr for his explanations of the workings of the UNIX system and device drivers, as well as Ted Kowalski for his help in debugging several stand-alone programs and his suggestions of practical extensions to that which already existed.

References

[1] Dolotta, T. A., Olsson, S. B., and Petruccelli, A. G., eds. *UNIX User's Manual*—Release 3.0. Bell Laboratories, June 1980.

[2] *UNIX Time-Sharing System: UNIX Programmer's Manual*—Seventh Edition. Bell Laboratories, January, 1979.

[3] *UNIX/32V Time-Sharing System: UNIX Programmer's Manual*—Version 1.0. Bell Laboratories, February, 1979.

[4] *Peripherals Handbook*. Digital Equipment Corporation, 1978.

[5] *PDP 11/70 Processor Handbook*. Digital Equipment Corporation, 1976.

[6] *VAX 11/780 Architecture Handbook*. Digital Equipment Corporation, 1977.

[7] *VAX 11/780 Hardware Handbook*. Digital Equipment Corporation, 1978.

Appendix: The Stand-Alone Command Interpreter

The stand-alone command interpreter is called `sash` (for *s*tand-*a*lone *sh*ell). It is a glorified UNIX boot program. `Sash` is begun running through whatever means available. It relocates itself up to high memory and executes there. When it is running, it prompts with `$$`.

`Sash` accepts three types of commands. The most common type is the *program execution* command. Here the user types the name of the stand-alone program to be executed, followed by arguments to be passed to the program. The program name and arguments are separated by spaces or tabs, and the single-quote and double-quote characters are properly understood (for arguments containing special characters within them). For example, if `/stand/ls` is a stand-alone program that does the same as the UNIX `ls` program, then in order to get a long listing of the contents of the directory `/tmp`, the user would type:

```
$$ /stand/ls -l /tmp
```

UNIX itself may be booted by using this method:

```
$$ /unix
```

The second type of command is the `cd` command. `Sash` has a notion of its *current directory*. All programs that are called with names that do not begin with a slash (`/`) are searched for relative to the current directory. When `sash` is begun executing, the current directory is the root directory (`/`). Thus, in the previous paragraph, UNIX could have been booted by typing:

```
$$ unix
```

If the `cd` command is invoked with an argument, then the argument becomes the current directory. The following sequence is equivalent to the `ls` command discussed above:

```
$$ cd /stand
$$ ls -l /tmp
```

The current directory is local to the `sash` program. It is remembered from one `sash` command to the next. It is not, however, passed on to the invoked program. Arguments that are passed to programs must therefore be relative to the root directory.

If `cd` is invoked with no arguments, the value of the current directory is printed.

`Sash` has a default notion of the disk type and unit number for the root file system, as well as for a `/usr` file system. These are generally slices 0 and 1, respectively, of unit 0 of the RP04/05/06 disk. (The defaults may be easily changed by recompiling the `sash` source.) To change `sash`'s ideas of disk type and unit number, the `set` command may be used. There are two basic forms of the `set` command: `set unit` and `set disk`. Their synopses are:

```
set unit {/|/usr} {0|1|...|7}
set disk {/|/usr} {rk05|rp03|rp04}
```

where `{...|...}` indicates a mandatory choice. On the VAX-11/780, the `rk05` and `rp03` choices do not exist. For example, in order to execute a stand-alone program in `/usr/steve/saprog` where `/usr` is the file system on slice 1 of RP03 unit 2, the user may type:

```
$$ set disk /usr rp03
$$ set unit /usr 2
$$ /usr/steve/saprog
```

The notions of disk type and unit number are, like current directory, local to `sash`, and are not passed to the invoked program, which has its own idea of where `/usr` (if any) and the root file system are located.

The UNIX Equipment Test Package: Operational Procedures

A. L. Chellis
T. J. Kowalski

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

The Equipment Test Package (ETP) is a collection of hardware exercisers that run on the UNIX† operating system. The hardware is exercised by using *shell* scripts and the operating system itself to generate a large number of reads and writes to all devices. The reads and writes test all combinations of I/O and devices under heavy load conditions.

The purpose of this document is to explain the procedures for running the ETP. It first presents a list of materials needed to use the ETP. The user is then shown how to place the ETP on disk and in memory. Finally, the procedures to run and reconfigure the ETP are explained.

2. CHECKLIST

In preparation for running the ETP, the user should have the following:

- A copy of the ETP tape marked "PDP-11" or "VAX" (as appropriate) and this document.
- A formatted, flag-free disk.
- Knowledge of the bootstrap loader program for disk drive and tape drive (the appendices to this document contain information on commonly used loaders).
- Knowledge of the hardware configuration for each machine:
  - — type of processor;
  - — K words of memory;
  - — existence of floating-point hardware;
  - — names, addresses, and vectors of all devices.

3. BOOT PROCEDURES

Booting the ETP consists of two distinct steps: initial load to disk and loading into memory. To prepare to run the ETP, you must produce a "bootable" ETP disk pack from the distribution tape. Then you must bring the ETP into memory by booting the ETP disk pack. Additionally, a special test is provided for PDP-11/70's to exercise their memory management registers.

3.1 Initial Load to Disk

The ETP is normally distributed on a single, multi-file magnetic tape, recorded in 9-track format at 800 bpi. The tape is marked either "PDP-11" or "VAX"; be sure you have the correct tape for your machine. The ETP is a disk-based system exerciser. Therefore, the ETP must be placed on disk before it can used. To place the ETP on disk:

---

† UNIX is a trademark of Bell Laboratories.

1. Mount the ETP distribution tape (*without* a write ring) at load point.
2. Boot the ETP tape:

   PDP-11 This tape boots in the same manner a DEC diagnostic tape. If you do not have a hardware bootstrap for the tape drive, see Appendix 2.

   VAX The floppy delivered with the VAX does not have tape-boot capability; see Appendix 3.
3. Follow the directions printed on the console. Samples of PDP-11 and VAX console dialogue can be found in Appendix 6.
4. When the tape rewinds, HALT the CPU.

3.1.1 Initial test of PDP-11/70 CPU memory management registers. The file */stand/mmtest* is a stand-alone diagnostic program for the PDP-11/70's memory management registers. It should be booted and run (20 minutes) if you are not *absolutely* sure that DEC FCO (field change order) M8140-R002 has been applied to your PDP-11/70. To place the diagnostic in memory, use the hardware bootstrap loader to boot the disk you have just created. The disk boots just like a DEC diagnostic. If you do not have a hardware bootstrap, see Appendix 2.

To start the memory management test, proceed as follows (note that "<NO CR>" means "do not hit carriage return"):

[sys] # <NO CR>

[you] 0 <NO CR>

[sys] = <NO CR>

[you] /stand/mmtest

The memory management test will begin to run; when it is complete, it will print "DONE" on the console terminal.

If any errors occur during this test, the ETP will *not* run until your hardware maintenance contractor applies FCO M8140-R002 to your PDP-11/70 CPU.

3.2 Booting ETP from the Disk

To place ETP in memory, boot the disk you have created using the procedures for the PDP-11 or VAX, as appropriate.

3.2.1 Boot procedures for PDP-11. Place the ETP in memory by booting the disk you have just created. The disk boots just like a DEC diagnostic. If you do not have a hardware bootstrap for the disk drive, see Appendix 2. Proceed as follows:

[sys] # <NO CR>

[you] 0 <NO CR>

[sys] = <NO CR>

[you] *name*

where *name* is the name printed out by the initial load-to-disk program.

3.2.2 Boot procedures for VAX. The floppy disk delivered with the VAX does not have UNIX disk-boot capability; see Appendix 3. Proceed as follows:

[sys] $$ <NO CR>

[you] *name*

where *name* is the name printed out by the initial load-to-disk program.

3.3 Common boot procedures for PDP-11 and VAX

Once the ETP is placed into memory, the running of the ETP is the same for PDP-11 and VAX:

[sys] UNIX/etp1.3: *name*
real mem = MMMM bytes
avail mem = NNNN bytes
enter date in the following format: MMddhhmmyy <NO CR>

[you] MMddhhmmyy

where *MM* is the month (01-12), *dd* is the day of the month (01-31), *hh* is the hour of the day (00-23), *mm* are the minutes past the hour (00-59), and *yy* are the last two digits of the year (70-??).

The current date will be echoed and the ETP will check the disk pack just generated and then initialize itself; this process takes about five minutes. See Appendix 6 for the console dialogue.

After the startup is complete, you should login as follows:

[sys] login: <NO CR>

[you] etp

The ETP will identify itself and print out its version number, e.g.:

Equipment Test Package Version 1.3

4. RUNNING THE ETP

The ETP is run in two parts: a general-purpose configuration and a specific configuration for your system. This is done to allow you the flexibility to reconfigure the ETP as your system's configuration changes. You may save your specific configuration once you've booted a configuration with a tape device.

4.1 Initial Test of Root Device.

Before generating specific configurations, the root device, memory, and CPU speed on which those configurations will reside must be tested. If any of these devices malfunction, it is useless to proceed further. To begin the tests, enter the command:

etpall [*loop_count*]

Where *loop_count* indicates how many times you wish to loop through the entire test. The default *loop_count* is 1. When all the tests have been completed and no errors have been detected, you are ready to generate other configurations. To begin the generation, enter the command:

etpgen

This interactive program will prompt for all the information it needs to generate an ETP for up to 4 configurations. It is limited to 4 because of disk-space limitations. The names used to describe a configuration are listed in Appendix 5 in the "Device" column. A sample run is shown in Appendix 6.

When the generation is complete, you must shut down the system. To shut down the system, enter the command:

shutdown

4.2 Tests for Specific Configurations

Repeat all of the steps shown in Section 3.2 above, *but instead of typing the* name *used in the initial boot, type the* project name *of the configuration you wish to test.* The "project name" (entered

as the first line of a configuration during generation) identifies a specific configuration. It must be only one word of no more than 14 characters.

If a tape device has been configured into the new system, you may save all generated configurations on tape. To save the configurations on tape, enter the command:

etptape

A bootable ETP tape will be created on drive 0 of the tape device configured into the system. This tape can be booted by the same procedures used for the original ETP distribution tape.

The ETP can be run in either interactive or non-interactive mode. Interactive mode is used to test character devices that require operator intervention. The default mode of operation is non-interactive.

To change or inquire about the mode of operation of the ETP, enter the command:

etpchmod [−i] [−n]

With no options, *etpchmod* prints the current status. The *−i* and *−n* options change the mode of testing to interactive and non-interactive, respectively.

5. TESTS

The ETP provides a number of individual tests. For ease of use, it also provides a test that performs all the individual tests. This section briefly describes this all-encompassing test and each individual test.

5.1 Test of All Devices

The all-encompassing test includes tests of all block and character devices, memory, CPU speed, and a load bus routine. To begin the test, enter the command:

etpall [*loop_count*] [*test* ...]

If individual *tests* are specified on the command line, only those tests will be performed. The test names are *block*, *char*, *mem*, *time*, and *load*. The length of time needed for this test is dependent upon the number of devices on the system and the size of memory.

5.2 Test of Block Devices

To test individual block devices, enter the command:

etpblock [*loop_count*] [*device* ...]

If individual *devices* are specified on the command line, only those devices will be tested. If no devices are specified, all of the block devices configured into the system at ETP generation time will be tested. The list of possible devices is given in Appendix 5 in the "Generic" column.

5.3 Test of Character Devices

To test individual character devices, enter the command

etpchar [*loop_count*] [*device* ...]

If individual *devices* are specified on the command line, only those devices will be tested. If no devices are specified, all of the character devices configured into the system at ETP generation time will be tested. The list of possible devices is given in Appendix 5. User interaction is required for testing the *dh*, *dz*, and *dn* lines.

5.4 Test of Memory and Swap Device

To test all of memory, floating-point hardware, and the swap device, enter the command:

etpmem [*loop_count*]

Memory and the swap device are filled with programs that store and retrieve test patterns of characters, integers, and double floating-point quantities.

5.5 Test of CPU Throughput

To test the throughput of a CPU, enter the command:

etptime [*loop_count*]

Throughput of a CPU may vary from machine to machine, because of variation in memory speeds, CPU speeds, and cache speeds. The numbers in Appendix 7 are rough guides as to what you should expect for your configuration.

5.6 Test of Bus Loading

To start simultaneous I/O on all devices, enter the command:

etpload [*loop_count*]

This test starts simultaneous I/O activity on all tape and disk devices, thereby loading the UNIBUS.

6. ERROR REPORT

The ETP logs all of the errors it detects in an error log file. This file is not removed when the system is shut down or booted up. In effect, new errors are logged at the end of the log file if this file already exists.

This error log can be printed out in a terse format. To print out all the errors logged to date, enter the command:

errpt −a

The resulting error report can be easily interpreted by your hardware maintenance contractor. For further information on the *errpt* command, see Appendix 4.

To save the current error log (which is file **/usr/adm/errfile**) in **/usr/adm/oerrfile**, enter the command:

etperrmv

This command will temporarily stop error logging, move the current error log file, and restart error logging.

7. ACKNOWLEDGEMENTS

We gratefully acknowledge the work of the originators of the Equipment Test Package, Rick Brandt and Cathy Perez. Special thanks are due to the hundreds of users who provided extensive feedback, helping us to make the package more effective and easier to use.

Appendix 1: PDP-11/70 Boot

The PDP-11/70 has a dedicated hardware bootstrap loader called the M9301-YC. This allows it to bootstrap programs from a wide range of storage media.

The M9301-YC attempts to boot from the device and drive number specified in the console switches. Console switches 7-3 select the device, while console switches 2-0 select the drive number. The table below below describes the devices selected for each switch setting.

To start operation of the bootstrap loader:

1. Halt the CPU by depressing the HALT switch.
2. Set the Address Display select switch to CONS PHY.
3. Set the Console Switch Register to 165 000 (octal).
4. Depress the LOAD ADRS switch.
5. Reset the console switches to 0.
6. Set switches 7-0 for the desired device.
7. Put the HALT switch in the ENABL position.
8. Depress the START switch.

The selected device will be booted. This takes approximately three seconds.

Any error during the boot will cause the CPU to halt. A list of possible halt addresses and their meanings is given in the DEC PDP-11/70 Processor Handbook in the chapter on Console Operation.

Console Switches (7-3)	*Device*	*Name*
00	illegal	—
01	TM11/TU10	Magnetic tape
02	TC11/TU56	DECtape
03	RK11/RK05	Disk pack
04	RP11/RP03	Disk pack
05	reserved	—
06	RH70/TU16	Magnetic tape
07	RH70/RP04	Disk pack
10	RH70/RS04	Fixed-head disk
11	RX11/RX01	Diskette
12-37	illegal	—

Appendix 2: PDP-11 ROM Boot

Standard DEC ROM bootstrap loaders may not correctly execute UNIX initial load programs. Therefore, special bootstrap loaders were designed that may be manually toggled into memory.

Each special bootstrap loader is position-independent, that is, it may be placed anywhere in memory. Normally, it is placed in high memory to avoid being overwritten. Each special bootstrap loader reads one block from drive 0 into memory starting at address 0 and then jumps to address 0. To minimize the size of the special bootstrap loaders, they each assume that a hardware INIT was generated prior to execution. In each case, each special bootstrap loader will read in at least 256 words, which is the maximum size of the UNIX initial load.

On disk devices, block 0 is read. On tape devices, one block starting at the current position of the tape is read, so that, the tape should normally be positioned at the load point prior to booting.

Below, we give the the octal listing of five such special bootstrap loaders, together with the corresponding assembly language instructions.

TU10 — Magnetic Tape:

```
012700          mov     $mtcma,r0
172526
010040          mov     r0,-(r0)        /use magnetic tape addr for byte count
012740          mov     $60003,-(r0)    /read, 800 bpi, 9 track
060003
105710  1:      tstb    (r0)            /wait for ready
002376          bge     1b
005007          clr     pc              /transfer to zero
```

TU16 — Magnetic Tape:

```
012700          mov     $mtwc,r0
172442
012760          mov     $1300,30(r0)    /set 800 bpi, PDP format
001300
000030
010010          mov     r0,(r0)         /use magnetic tape addr for word count
012740          mov     $71,-(r0)       /read
000071
105710  1:      tstb    (r0)            /wait for ready
002376          bge     1b
005007          clr     pc              /transfer to zero
```

RK05 — Disk Pack:

```
012700          mov     $rkda,r0
177412
005040          clr     -(r0)
010040          mov     r0,-(r0)        /use RK05 addr for word count
012740          mov     $5,-(r0)        /read
000005
105710  1:      tstb    (r0)            /wait for ready
002376          bge     1b
005007          clr     pc              /transfer to zero
```

RP03 – Disk Pack:

```
012700          mov     $rpmr,r0
176726
005040          clr     -(r0)
005040          clr     -(r0)
005040          clr     -(r0)
010040          mov     r0,-(r0)        /use RP03 addr for word count
012740          mov     $5,-(r0)        /read
000005
105710    1:    tstb    (r0)            /wait for ready
002376          bge     1b
005007          clr     pc              /transfer to zero
```

RP04 – Disk Pack:

```
012700          mov     $rpcs1,r0
176700
012720          mov     $21,(r0)+       /read-in preset
000021
012760          mov     $10000,30(r0)   /set to 16-bits/word
010000
000030
010010          mov     r0,(r0)         /use RP04 addr for word count
012740          mov     $71,-(r0)       /read
000071
105710    1:    tstb    (r0)            /wait for ready
002376          bge     1b
005007          clr     pc              /transfer to zero
```

Appendix 3: VAX-11/780 Boots

1. TAPE BOOT

The floppy disk delivered with the VAX does not have UNIX tape-boot capability. The user must type in the following program to read the first record on tape drive 0 (type a carriage return at the end of each input line):

```
>>> H
>>> U
>>> I

        INIT SEQ DONE

>>> D 20000 20008FD0
>>> D+ D0502001
>>> D+ 3204A001
>>> D+ C003C08F
>>> D+ A0D40424
>>> D+ 8FD00C
>>> D+ C0800000
>>> D+ 8F320800
>>> D+ 10A0FE00
>>> D+ C007D0
>>> D+ C039D004
>>> D+ 400
>>> S 20000                (Starts tape load)

        HALT INST EXECUTED
        HALTED AT 0002002F

>>> S 0                    (Execute boot program loaded from tape)
```

From this point on, the loader initiates a question-and-answer sequence to control the remainder of the load process.

2. DISK BOOT

The floppy disk delivered with the VAX does not have UNIX disk-boot capability. The user must type in the following program to read the first block on disk drive 0 (type carriage return at the end of each line):

```
>>>H
>>>LINK                    (Save the following sequence on the floppy)
                           (the prompt should change to "<<<")
<<<H
<<<U
<<<I
<<<D 20000 00009FDE       (Boot program for MBA 0, drive 0)
<<<D+ D0512001
<<<D+ D004A101
<<<D+ 0400C113
<<<D+ 10008F32
<<<D+ D40424C1
<<<D+ 8FD00CA1
<<<D+ 80000000
<<<D+ 320800C1
<<<D+ A1FE008F
<<<D+ 28C1D410
<<<D+ 14C1D404
<<<D+ C139D004
<<<D+ 400
<<<S 20000
<<<S 2
<<<Control-C              (Exit LINK load)
>>>
```

You are now ready to boot UNIX. Each time it is necessary to boot (or reboot) UNIX, one simply follows the sequence:

```
>>>P                      (This executes the commands saved in the floppy link file;
                           the console should echo each command in the file.)
$$ unix<cr>               (Load and execute /unix)
```

Appendix 4: Error Report

The following command may be used print out various aspects of the error log file:

errpt [**−a**] [*−dev ...*] [**−int**] [**−mem**] [**−s***date*] [**−e***date*] [**−p***n*] [**−f**] [*file ...*]

Errpt processes data collected by the error logging mechanism (*errdemon*(1M) entry in the *UNIX User's Manual*) and generates a report of that data. The default report is a summary of all errors posted in the named files. Options apply to all files and are described below. If no files are specified, *errpt* attempts to use **/usr/adm/errfile** as *file*.

A summary report indicates the options that may limit its completeness, gives the times of the earliest and latest errors encountered, and gives the total number of errors of one or more types. Each device summary contains the total number of unrecovered errors, recovered errors, errors unable to be logged, I/O operations on the device, and miscellaneous activities that occurred on the device. The number of times that *errpt* has difficulty reading input data is included as read errors.

A detailed report contains, in addition to specific error information, all instances of the error logging process being started and stopped and any time changes (via *date*(1)) and configuration changes (for UNIX/RT only) that took place during the interval being processed. A summary of each error type included in the report is appended to a detailed report.

A report may be limited to certain records in the following ways:

−s*date* Ignore all records posted earlier than *date*, where *date* has the form **MMddhhmmyy**, as for the *date*(1) command.

−e*date* Ignore all records posted later than *date*.

−a Produce a detailed report that includes all error types.

−dev Limit a detailed report to *dev*, a block device identifier. *Errpt* is familiar with the common form of identifiers. Currently, the block devices for which errors are logged are RP03, RP04, RP05, RP06, RS03, RS04, TU10, TU16, RK05, RF11, RL01.

−int Include in a detailed report errors of the stray-interrupt type.

−mem Include in a detailed report errors of the memory-parity type.

−p*n* Limit the size of a detailed report to *n* pages.

−f In a detailed report, limit the reporting of block device errors to unrecovered errors.

Appendix 5: Generic Names for Peripheral Devices

Processors

The Equipment Test Package is currently available for the following processors:

PDP-11/70, 11/45, 11/34
VAX 11/780

Devices

There are testing procedures defined for the following devices. The *Device names* are used when entering a configuration during generation. The *Generic names* are used when running the tests.

Device Name	*Generic Name*
dh11	dh
dm11	dm
dn11	dn
dz11	dz
dzkmc	dzk
kmc11	kmc
lp11	lp
rf11	rf
rk05	rk
rl01, rl11	rl
rp03, rp11	rp
rp04, rp05, rp06	hp
rs04, rs03	hs
tu10, tm11	tm
tu45, tu77, tu16, te16	ht
vp	vp

There are *no* testing procedures defined for the following devices, but they may be entered into a configuration so that they may be accessed by the user.

Device Name	*Generic Name*
da11b	da
dl11, la36, kl11	kl
dmc11	dmc
dqs11b, dqs11a	dqs
dr11c	cat
du11	du

Appendix 6: Sample Run

Boot Procedures for the PDP-11*

UNIX tape boot loader

Equipment Test Package Version 1.3

Initial Load: Tape-to-Disk

The disk drive type which will be used for the Root file system
and the tape drive type which will be used for the Initial Load Tape
must be specified below.

Answer the questions with a 'y' or 'n' followed by
a carriage return or line-feed.
There is no type-ahead — — — wait for each question to complete.
The character '@' will kill the entire line
and the character '#' will erase the last character typed.
To restart the program during the question phase,
type the DEL character.

PDP-11/70?: **y**

RP03 at address 176710?: **n**
RP04/5/6 at address 176700?: **y**
Drive number (0-7)?: **0**
Disk drive 0 selected

Mount formatted pack on drive 0
Ready?: **y**

TU10/TM11 at address 172520?: **n**
TU16 at address 172440?: **y**
Drive number (0-7)?: **0**
Tape drive 0 selected

The tape on drive 0 will be read from the current position
at 800bpi, 5120 characters (10 blocks) per record
and written onto the pack on drive 0 starting at block 0.

Ready?: **y**
Size of file system to be copied is 4000 blocks.
The pack will be labeled etp1.3;
disk boot block for your disk drive type will be installed now.

The file system copy is now completed.

To boot the basic ETP for your disk
as indicated above, mount this pack on drive 0
and read in the boot block (block 0) using
whatever means you have available. See Appendix 1 in
Equipment Test Package: Operational Procedures.

Then boot the program hp.
Normally: **#0=hp**

---

* User's responses are shown in **bold**.

ETP will come up and ask you for the date and ask you to login. Please see Equipment Test Package: Operational Procedures for further details.

Good Luck!

The tape will now be rewound.

Boot Procedures for the VAX-11/780

UNIX tape boot loader

Equipment Test Package Version 1.3

Initial Load: Tape-to-Disk

The disk drive type which will be used for the Root file system
and the tape drive type which will be used for the Initial Load Tape
must be specified below.

Answer the questions with a 'y' or 'n' followed by
a carriage return or line-feed.
There is no type-ahead — — — wait for each question to complete.
The character '@' will kill the entire line
and the character '#' will erase the last character typed.
To restart the program during the question phase,
type the DEL character.

VAX-11/780?: **y**

RP06 at NEXUS 8?: **y**
Drive number (0-7)?: **0**
Disk drive 0 selected

Mount formatted pack on drive 0
Ready?: **y**

TE16 at NEXUS 9?: **y**
Drive number (0-7)?: **0**
Tape drive 0 selected

The tape on drive 0 will be read from the current position
at 800bpi, 5120 characters (10 blocks) per record
and written onto the pack on drive 0 starting at block 0.

Ready?: **y**
Size of file system to be copied is 6000 blocks.
The pack will be labeled etp1.3;
disk boot block for your disk drive type will be installed now.

The file system copy is now completed.

To boot the basic ETP for your disk
as indicated above, mount this pack on drive 0
and read in the boot block (block 0) using
whatever means you have available. See Appendix 3 in
Equipment Test Package: Operational Procedures.

Then boot the program hp.
Normally: **$$ hp**

ETP will come up and ask you for the date and ask you
to login. Please see Equipment Test Package: Operational
Procedures for further details.

Good Luck!

The tape will now be rewound.

Common Boot Procedures for the PDP-11 and VAX-11/780

```
UNIX/etp1.3: hp
real mem  = 1048576
avail mem = 921088
enter date in the following format: MMddhhmmyy 0107113880
Mon Jan  7 11:38:53 EST 1980

                *** Equipment Test Package Start for Project: hp ***

Check Root Filesystem

/dev/hp0
File System: master Volume: etp1.3

** Phase 1 - Check Blocks and Sizes
** Phase 2 - Check Pathnames
** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Counts
** Phase 5 - Check Free List
303 files  3151 blocks  2660 free

ETP Start Complete

login: etp
Equipment Test Package System - Version 1.3
```

Initial Test of Root Device

```
# etpall

                *** Equipment Test Package ***

*** Equipment Test Package Pass Number: 1       Jan 07 11:39

                *** Block Device Tests ***

*** Block Test Pass Number: 1   Jan 07 11:39

Testing null with hp0a   Jan 07 11:39

Copy 2000 records of size 512 bytes from hp0a to null
2000+0 records in
2000+0 records out
Copy 1000 records of size 1024 bytes from rhp0a to null
1000+0 records in
1000+0 records out
Copy 100 records of size 10240 bytes from rhp0a to null
100+0 records in
100+0 records out
Copy 50 records of size 20480 bytes from rhp0a to null
50+0 records in
50+0 records out

Testing hp0a with hp0a   Jan 07 11:40

Making filesystem on hp0a
507 blocks
Checking filesystem on hp0a
```

```
Copy 2000 records of size 512 bytes from hp0a to hp0a
2000+0 records in
2000+0 records out
Copy 1000 records of size 1024 bytes from rhp0a to rhp0a
1000+0 records in
1000+0 records out
Copy 100 records of size 10240 bytes from rhp0a to rhp0a
100+0 records in
100+0 records out
Copy 50 records of size 20480 bytes from rhp0a to rhp0a
50+0 records in
50+0 records out

Testing hp0b with hp0a   Jan 07 11:43
   :
   :
Block Device Tests Complete

                                   *** CPU Timing Test ***

*** CPU Timing Test Pass Number: 1      Jan 07 11:55

CPU time:      14.2

Compare the CPU time with those in Appendix 7

CPU Timing Test Complete

                                *** Character Device Tests ***

                                 ** Non-interactive Mode **

*** Character Test Pass Number: 1       Jan 07 11:55

Character Device Tests Complete

                                   *** I/O Bus Load Test ***

*** I/O Bus Load Pass Number: 1   Jan 07 11:55

I/O Bus Load Test Complete

                               *** Memory and Swap Device Test ***

*** Memory and Swap Test Pass Number: 1   Jan 07 12:02

Memory and Swap Test Complete

                                    Summary Error Report

  Summary Error Report       Prepared on Jan  7 12:06          Page  1

       Error Types: All
       Limitations:
       Date of Earliest Entry: Mon Jan  7 11:38:55 1980
       Date of Latest   Entry: Mon Jan  7 11:40:02 1980

            Total Stray Interrupts          -  0
            Total Memory Parity Errors      -  0

ETP Complete
#
```

Generation For Specific Configurations

 # etpgen
Equipment Test Package (ETP) Generation

Please enter system configuration.
You will be in the editor.
To begin, you must enter:
a
When finished, you must enter:
.
w
q
Do you want to see format rules? (y or n) **y**

FORMAT:

* project name
* K—words of core
* type of processor (vax)
* floating point or not (fpp, nfpp)
device (tab) vector (tab) address (tab) number-of-devices

NOTE:
1. The project name must be only one word of no more than 14 characters.
2. The device names should be selected from the DEVICE column in Appendix 5 of the "Operational Procedures" manual.
3. If the number of devices is omitted, a maximum number will be assumed, so be careful when entering numbers.
4. List each dh, dm, dz, and dn as separate entries, and leave the number-of-devices column blank, unless there are less than:

   16 lines/device on each: dh, dm
   8 lines/device on each: dz
   4 lines/device on each: dn

Proceed to enter system configuration:

a
*** vaxe**
*** 512**
*** vax**
*** fpp**
rp06 0 0 2
te16 0 0 1
dn11 310 775200
dz11 320 760100
dz11 550 760120
kmc11 300 760070
.
w
q

```
Any more projects to be on the same disk/tape? (y or n)  n
Building Rest of Configuration File for: vaxe
Taking Care of Necessary Devices for: vaxe
Making Operating System for: vaxe

ETP Generation Complete
#
# shutdown

SHUTDOWN PROGRAM

Mon Jan  7 13:56:02 EST 1980

  NOTE:
       If this command has not completed in 10 minutes, do the following:
       1) Hit the DEL key
       2) Execute the following commands:
          killall
          sync
          init 1
          fsck
       3) Halt the system

All currently running processes will now be terminated

   PID   TTY   TIME   COMMAND
     0     ?   3:54   swapper
     1     ?   0:00   init
    45    co   0:01   sh
   323    co   0:00   ps
   291    co   0:01   sh

HALT the system

#
```

Tests for Specific Configurations

$$ vaxe

UNIX/etp1.3: vaxe
real mem = 1048576
avail mem = 916992
enter date in the following format: MMddhhmmyy **0107140280**
Mon Jan 7 14:02:00 EST 1980

*** Equipment Test Package Start for Project: vaxe ***

Check Root Filesystem

/dev/hp0
File System: master Volume: etp1.3

** Phase 1 - Check Blocks and Sizes
** Phase 2 - Check Pathnames
** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Counts
** Phase 5 - Check Free List
303 files 3151 blocks 2660 free

ETP Start Complete

login: **etp**
Equipment Test Package System - Version 1.3
#
etpchmod
Non-interactive
etpchmod -i

etpchar dn dz[0-7] kmc

*** Character Device Tests ***

** Interactive Mode **

*** Character Test Pass Number: 1 Jan 07 14:05

DN TEST

Please enter the phone # of a telephone in this room: **1234**
When that phone rings,
pick up the receiver to establish communication and then replace it.
That number will be dialed as many times as there are dn lines to be tested.

Testing DN Line: dn0
Testing DN Line: dn1
Testing DN Line: dn2
Testing DN Line: dn3

KMC TEST

Testing kmc0: kmc0 okay

MULTIPLEXER TEST

Ready to test multiplexer lines at 300 baud
To do so:

a) log onto a terminal by dialing the phone number associated with each line
b) log in as 'tty'
c) You will automatically be logged off when the line has been tested

When finished testing lines, hit 'carriage return'.
Lines Tested: (in order of testing)

dz0
dz1
dz2
dz3
dz4
dz5
dz6
dz7

Character Device Tests Complete
#

Appendix 7: CPU Timings

Cache	*VAX 11/780*	*PDP 11/70*	*PDP 11/45*	*PDP 11/34*	*PDP 11/23*
on	14.5	16.6	22.8	33.9	N/A*
off	40.3	51.8	40.1	57.7	59.9

* Not available.

Appendix 8: Error Conditions

This Appendix contains a list all of the error messages produced by the set of programs that make up the ETP. Each error message has an error code, which may be used to refer to this Appendix. Below each error message, a possible cause and a related action are described. This Appendix is to be used only as a guide to probable causes and probable solutions. If any error persists, first make sure that the directions in this document were followed precisely. Other errors messages that may appear on the ETP printout are caused by the operating system itself, and usually merit some attention (see Appendix 4).

err001: usage: etpstart *project*

cause: A *project* name is missing as the first argument to the start-up procedure that is executed automatically upon bringing up the operating system. This error shows up when the file system has been corrupted.
action: Reboot the ETP using a new disk.

err002: unknown project: *project*

cause: A corrupted file system may have caused either the */project* or the **/usr/lib/etp/configs/***project* directory to be destroyed.
action: Regenerate the ETP configuration for that project.

err003: missing directory

cause: A file system directory that is required for execution of either the start-up or the project generation procedure is missing. This error may be caused by a corrupted file system.
action: Regenerate the ETP configuration for that project.

err004: no test devices for *project*

cause: None of the devices entered into the configuration for *project* at generation time are supported by the ETP.
action: Check the ETP configuration for *project* and regenerate the ETP configuration for that *project*.

err005: root unknown

cause: A corrupted file system may have caused the file **/usr/lib/etp/configs/running/root** to be destroyed or not created properly.
action: Regenerate the ETP configuration for that project.

err006: premature termination

cause: The currently running test procedure has been prematurely terminated because the user hit either the DEL key or the BREAK key.
action: A summary error report will be printed automatically. If you wish to stop it, hit the DEL key again.

err007: etpstart did not run correctly

cause: The start-up procedure, which is executed automatically upon bringing up the operating system, did not run properly.
action: Reboot the ETP.

err008: not configured for device: *device*

cause: The *device* argument typed in on the command line was not configured into the system during the generation of ETP for this project.
action: Make sure the generic name was used on the command line and check the configuration to make sure that the device was configured into the system for this project.

err009: no test for device: *device*

cause: The *device* specified is not supported by the ETP.
action: The *device* will available for use, but it will not be tested by the ETP.

err010: conf file is missing

cause: Error in the generation procedure.
action: Reissue the **etpgen** command.

err011: illegal root device: *device*

cause: The ETP will not fit on *device*, which is the largest capacity disk entered into the configuration.
action: ETP needs at least one disk with at least 4000 blocks for PDP-11 systems and 6000 blocks for VAX systems. If the system configuration for the project does not include such a disk, the ETP cannot run on that system.

err012: no disk devices in system configuration

cause: No supported disk devices were entered into the configuration for the project.
action: The ETP requires at least one supported disk in a system.

err013: invalid processor type: *processor*

cause: The *processor* used in the configuration for the project is not supported by ETP.
action: Make sure that the processor type was entered correctly.

err014: $proj must be exported to this shell

cause: Parameters are not being placed in the environment correctly.
action: Reboot the ETP.

err015: ETP Generation Failed

cause: The configuration entered for the project is bad.
action: Check the configuration for the project to make sure that it was entered correctly and regenerate the ETP configuration for that project.

err016: there is no etp*test* test available

cause: The *test* specified on the command line does not exist.
action: Refer to the body of this document for usage and available tests.

UNIX Implementation

K. Thompson

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

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

1. INTRODUCTION

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

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

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

2. PROCESS CONTROL

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

The user process may execute from a read-only text segment, which is shared by all processes executing the same code. There is no *functional* benefit from shared-text segments. An *efficiency* benefit comes from the fact that there is no need to swap read-only segments out because the original copy on secondary memory is still current. This is a great benefit to interactive programs that tend to be swapped while waiting for terminal input. Furthermore, if two processes are executing simultaneously from the same copy of a read-only segment, only one copy needs to reside in primary memory. This is a secondary effect, because simultaneous

† UNIX is a trademark of Bell Laboratories.

execution of a program is not common. It is ironic that this effect, which reduces the use of primary memory, only comes into play when there is an overabundance of primary memory, that is, when there is enough memory to keep waiting processes loaded.

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

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

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

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

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

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

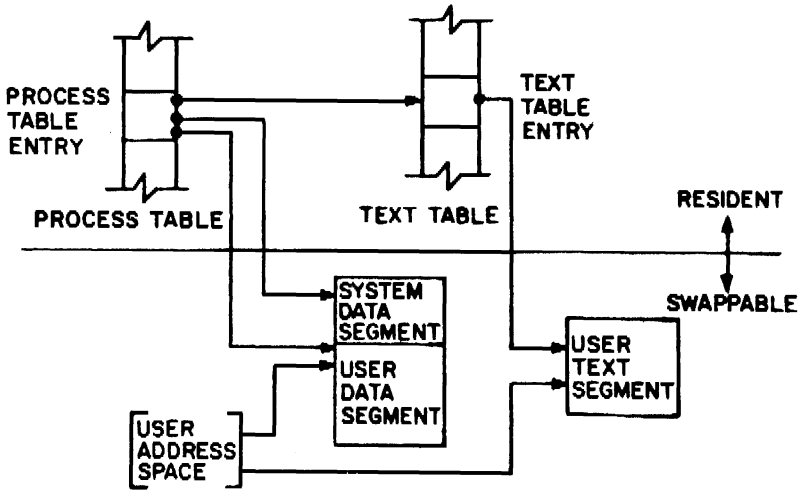


Fig. 1—Process Control Data Structure

2.1. Process creation and program execution

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

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

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

2.2. Swapping

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

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

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

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

2.3. Synchronization and scheduling

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

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

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

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

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

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

3. I/O SYSTEM

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

I/O" is a complete misnomer.

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

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

3.1. Block I/O system

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

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

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

3.2. Character I/O system

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

3.2.1. Disk drivers

Disk drivers are implemented with a queue of transaction records. Each record holds a read/write flag, a primary memory address, a secondary memory address, and a transfer byte count. Swapping is accomplished by passing such a record to the swapping device driver. The block I/O interface is implemented by passing such records with requests to fill and empty system buffers. The character I/O interface to the disk drivers create a transaction record that points directly into the user area. The routine that creates this record also insures that the user is not swapped during this I/O transaction. Thus by implementing the general disk driver, it is possible to use the disk as a block device, a character device, and a swap device. The only really disk-specific code in normal disk drivers is the pre-sort of transactions to minimize latency for a particular device, and the actual issuing of the I/O request.

3.2.2. Character lists

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

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

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

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

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

3.2.3. Other character devices

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

4. THE FILE SYSTEM

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

The UNIX file system is a disk data structure accessed completely through the block I/O system. As stated before, the canonical view of a "disk" is a randomly addressable array of 512-byte blocks. A file system breaks the disk into four self-identifying regions. The first block (address 0) is unused by the file system. It is left aside for booting procedures. The second block (address 1) contains the so-called "super-block." This block, among other things, contains the size of the disk and the boundaries of the other regions. Next comes the i-list, a list of file definitions. Each file definition is a 64-byte structure, called an i-node. The offset of a particular i-node within the i-list is called its i-number. The combination of device name (major and minor numbers) and i-number serves to uniquely name a particular file. After the i-list, and to the end of the disk, come free storage blocks that are available for the contents of files.

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

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

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

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

4.1. File system implementation

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

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

The user process accesses the file system with certain primitives. The most common of these are **open**, **create**, **read**, **write**, **seek**, and **close**. The data structures maintained are shown in Fig. 2.

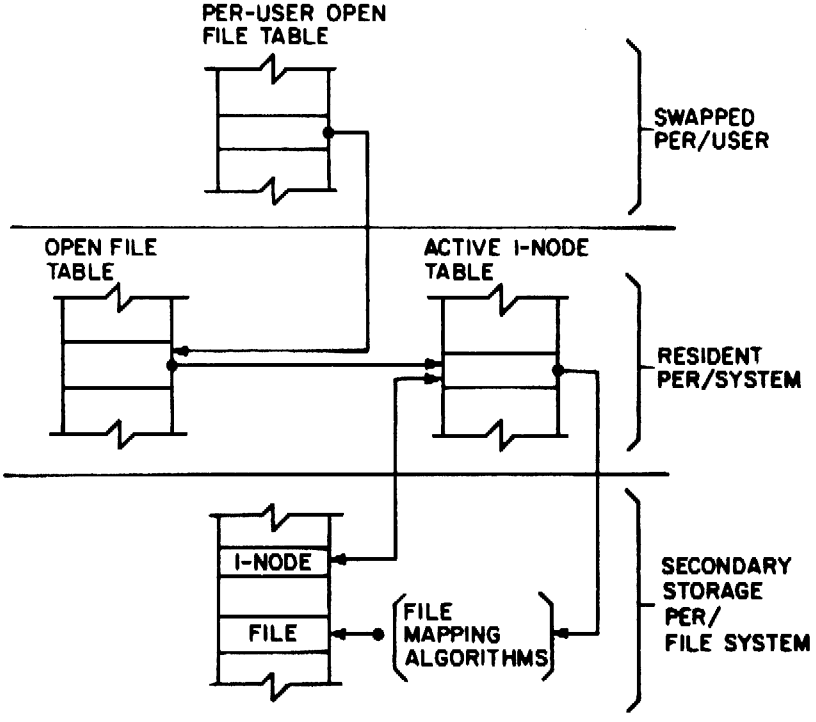


Fig. 2—File System Data Structure

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

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

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

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

4.2. Mounted file systems

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

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

4.3. Other system functions

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

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

have his own command language. Maintenance of such code is as easy as maintaining user code. The idea of implementing "system" code with general user primitives comes directly from MULTICS.[6]

5. REFERENCES

[1] R. E. Griswold and D. R. Hanson. An Overview of SL5, *SIGPLAN Notices* **12**(4):40-50 (April 1977).

[2] E. W. Dijkstra. Cooperating Sequential Processes, in *Programming Languages*, F. Genuys, ed., pp. 43-112, Academic Press (1968).

[3] J. A. Hawley and W. B. Meyer. *MUNIX, A Multiprocessing Version of UNIX*, M.S. Thesis, Naval Postgraduate School, Monterey, CA (1975).

[4] D. M. Ritchie and K. Thompson. *The UNIX Time-Sharing System*, Bell Sys. Tech. J. **7**(6):1905-29 (July-August 1978, Part 2).

[5] S. R. Bourne. *UNIX Time-Sharing System: The UNIX Shell*, Bell Sys. Tech. J. **7**(6):1971-90 (July-August 1978, Part 2).

[6] E. I. Organick. *The MULTICS System*, M.I.T. Press, Cambridge, MA (1972).

January 1981

The UNIX I/O System

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

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

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

Device Classes

There are two classes of device: *block* and *character*. The block interface is suitable for devices like disks, tapes, and DECtape which work, or can work, with addressable 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

† UNIX is a trademark of Bell Laboratories.

from *forks* after the file is opened. A *file* table entry contains flags which indicate whether the file was open for reading or writing or is a pipe, and a count which is used to decide when all processes using the entry have terminated or closed the file (so the entry can be abandoned). There is also a 32-bit file offset which is used to indicate where in the file the 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 device'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 stratagem avoids the overhead of having to call *wakeup* every time a buffer is freed on the chance that someone might want it.

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

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

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

Block Device Drivers

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

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

Although the most usual argument to the strategy routines is a genuine buffer header allocated as discussed above, all that is actually required is that the argument be a pointer to a place containing the appropriate information. For example the *swap* routine, which manages movement of core images to and from the swapping device, uses the strategy routine for this

device. Care has to be taken that no extraneous bits get turned on in the flag word.

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

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

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

Raw Block-Device I/O

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

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

UNIX on the PDP-11/23 and 11/34 Computers

T. J. Kowalski

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

During the past few years, the use of mini/micro computers in networks and small laboratory systems has steadily increased. Currently, the UNIX† operating system, used throughout the Bell System, is running primarily on DEC PDP-11/70s. With the advent of inexpensive computers similar in architecture to the PDP-11/70, in particular the PDP-11/23 microcomputer and the PDP-11/34 minicomputer, it became important that UNIX be available for these systems. The author set out, in June of 1978, to move UNIX from the PDP-11/70 to the PDP-11/34. The first version of UNIX on a PDP-11/34 with RL01 disk drives ran in July of 1978.

This paper describes architectural differences between the PDP-11/70 and the PDP-11/34 hardware,[1] their interaction with the UNIX operating system, and the changes the author implemented in that system to make it run on the PDP-11/34, along with some considerations for the future.

2. ARCHITECTURAL DIFFERENCES THAT AFFECT UNIX

There are many architectural differences between the PDP-11/70 and the PDP-11/34. For our purposes, it is important to understand only the differences that affect UNIX. The memory-management (MM) system, the availability of an instruction-backup register, the availability of additional register sets, the program-interrupt request register, and the set-priority-level instruction are all differences that affect the implementation of UNIX.

2.1 Memory Management

The PDP-11 family of computers is based upon a sixteen-bit virtual-address architecture. This architecture is implemented using pairs of MM registers. Each pair is composed of a MM-address register (containing the base physical address for mapping) and a MM-page-descriptor register (containing the length in bytes to be mapped and the direction of page expansion). The virtual address is mapped into a physical address by choosing a MM-address register and adding its contents times 0100 (octal) to the thirteen low-order bits of the virtual address. Thus, a pair of MM registers can map 8K bytes of virtual memory into 8K bytes of physical memory. The MM-address register is chosen by considering the current CPU mode (kernel, user, or supervisor), the type of memory reference (instruction space or data space), and the three high-order bits of the virtual address.

The PDP-11/70 has three CPU modes: *kernel* mode (K-mode), *user* mode (U-mode), and *supervisor* mode (S-mode). Each of these modes allows two types of reference: *instruction* space and *data* space.[2] Each type of reference has eight pairs of MM registers. Thus, the PDP-11/70 has sixteen pairs each of K-mode, U-mode, and S-mode MM registers. These 48 pairs of MM registers enable the PDP-11/70 to access 384K bytes of physical memory.

The PDP-11/34, on the other hand, has only two CPU modes: K-mode and U-mode. Each of these modes allows only one type of reference (instruction space), which has eight pairs of MM registers. Thus, the PDP-11/34 has eight pairs each of K-mode and U-mode MM registers.

---

† UNIX is a trademark of Bell Laboratories.

1. Unless explicitly stated otherwise, all references to the PDP-11/34 can be also applied to the PDP-11/23.
2. Instruction space is used for all instruction fetches, index words, absolute addresses, and immediate operands. Data space is used for all other references.

These 16 pairs of MM registers enable the PDP-11/34 to access 128K bytes of physical memory.

The absence of the data-space reference type on the PDP-11/34 requires all references to be mapped through the instruction space. This reduces the total amount of virtual memory per CPU mode from 128K bytes on the PDP-11/70 to 64K bytes on the PDP-11/34. This is by far the most serious restriction in moving the operating system and user programs from the PDP-11/70 to the PDP-11/34.

In many instances, the operating system moves data from the user to itself and vice versa. This requires the operating system to access physical memory not currently mapped by its K-mode MM registers. To accomplish this, the UNIX operating system must temporarily use MM registers belonging to another CPU mode. In the PDP-11/70, the S-mode is not used by UNIX. Therefore, the PDP-11/70 operating system uses its S-mode MM registers for this temporary addressability. Because the PDP-11/34 lacks a S-mode, the PDP-11/34 operating system must use its U-mode MM registers. This difference requires additional U-mode MM register saving and restoring in the PDP-11/34. This is a disadvantage both in CPU time and in the kernel space taken up by the code that saves these registers.

2.2 Instruction Backup

Temporary variables in a user program are stored in a last-in first-out data structure, which is called a *stack*. A user program in UNIX is run with an initial stack size of 768 bytes, which is expandable in 768 byte increments. The operating system will attempt to increase the stack size when it receives a MM trap from U-mode. After increasing the stack size, the program counter must be backed up and the instruction that caused the MM trap restarted. Unfortunately, some of the addressing modes of the PDP-11 have side effects that affect the general-purpose registers. These addressing modes are auto-increment/decrement of the general-purpose registers, and explicit references through the program counter. Thus, to restart an instruction, these side effects must be undone. On the PDP-11/70, there is a MM register, MMR1, that records any side effects on the general-purpose registers during execution of instructions. This register is used to reset the registers prior to restarting the instruction. The lack of this register for the PDP-11/34 forces a simulation of the source and destination addressing modes of the instruction that caused the MM trap. The lack of this register is very expensive in terms of kernel space taken up by the code that does this simulation.

2.3 Additional Set of General-Purpose Registers

The PDP-11/70 has a set of six additional general-purpose registers. Several critical UNIX routines run with interrupts disabled and utilize this set of registers. Because the PDP-11/34 lacks this set of registers, the PDP-11/34 UNIX must also use its registers for this purpose. This requires additional register saving and restoring, requiring more code in the kernel and more CPU time.

2.4 Program-Interrupt-Request Register

The PDP-11/70 has a program-interrupt-request register. This register is used to detect when the CPU is running at a priority level lower than or equal to a predefined priority level.

The UNIX operating system's algorithm for power-fail recovery depends on reaching a quiescent state[3] before processing the power-fail I/O recovery algorithm. This is accomplished by setting the program-interrupt-request register to interrupt when the CPU reaches priority level 1. The lack of this register in the PDP-11/34 forces its simulation when UNIX returns from interrupts. Because the UNIX operating system processes a great deal of interrupts, even a small amount of

3. Meaning that all interrupt processing started before the power-fail trap must be completed.

additional CPU time per interrupt is very costly.

2.5 Set-Priority-Level Instruction

Each time the UNIX operating system enters and exits critical pieces of kernel code, the CPU priority level is changed. The PDP-11/70 operating system uses the set-priority-level instruction (SPL). The lack of this instruction causes the PDP-11/34 to use a combination of bit-set and bit-clear instructions upon the processor status word (PSW). Because the UNIX operating system changes CPU priority levels a great deal, even a small amount of additional time per priority level change is also very costly.

3. IMPLEMENTATION OF PDP-11/34 UNIX

Moving UNIX from the PDP-11/70 to the PDP-11/34 required the author to write the machine-language assist functions for the PDP-11/34; these functions are written in the assembly language of the target computer to perform the following tasks:

- fault handling;
- memory management;
- speed-critical I/O and arithmetic operations;
- stack frame manipulation;
- hardware priority setting;
- register save and restore;
- machine-interrupt call to C procedure;

The PDP-11/34 machine-language assist functions are written with the same calling conventions as the PDP-11/70 machine-language assist functions and return the same values. This allows the same UNIX C language functions to be used on the PDP-11/70 and the PDP-11/34. In writing the PDP-11/34 machine-language assist functions, the author chose to partition the functions into separate files, as opposed to keeping the traditional *mch.s* file. This organization simplifies the management of the source code.

The module partitions and the algorithms used to handle the architectural differences described in Section 2 above are discussed in this section.

3.1 Module Names

The modules are subdivided by function. All *defines* are in *mch.h*, all the storage declarations are in *end.s*. The modules are listed below alphabetically, with a brief description of their function:

backup.s	attempt to back up an instruction that was only partially executed due to a MM trap from U-mode.
bufio.s	read and write of byte, integers, and longs in physical memory.
clist.s	*put* and *get* functions for the *cblock* structure.
copy.s	read and write large blocks of memory from virtual addresses to physical addresses, copy 64 bytes, clear 64 bytes, read and write from K-mode virtual addresses to U-mode virtual addresses.
csubr.s	save and restore registers that maintain the C stack frame.
cswitch.s	save and restore the user's registers and switch the operating system's idea of who is the currently-running user.
end.s	storage declarations.
fpp.s	save and restore the double-floating-point registers and status word.
math.s	long division, long remainder, minimum, and maximum functions.
mch.h	header file containing constants and definitions.
misc.s	process accounting and set-CPU-priority level.
power.s	save the state of the machine on a loss-of-power interrupt and restore the state of the machine on a resumption-of-power interrupt.

start.s	initialize MM registers, clear storage, and call main.
trap.s	all the fault handlers and the machine interrupt call to C procedure.
userio.s	read and write words and bytes in user's virtual addresses.

3.2 General Algorithms

The detailed description of the algorithms used to provide the functions necessary for the machine-language assist functions is divided into four categories. The algorithms manipulate the MM system, simulate the MMR1 register, simulate the program-interrupt-request register, and simulate the set-priority-level instruction.

3.2.1 Memory Management. In many instances, the operating system needs to access physical memory not currently mapped by the K-mode MM registers. The algorithm used to read and write physical memory utilizes the U-mode MM registers and the "move from/to previous instruction space" (MFPI/MTPI) instruction. To free the U-mode MM registers the old values must be saved on the kernel stack along with the current PSW; then the previous CPU mode (indicated by the PSW) must be set to U-mode. The long physical address is loaded into a pair of general-purpose registers and shifted ten bits to the left. The sixteen high-order bits of the result are the base physical address in *core clicks*[4] for the virtual address. The base address is loaded into a U-mode MM-address register and a 077406[5] (octal) is loaded into the corresponding MM-page-descriptor register. The sixteen high-order bits are cleared with the exception of bits 9-7, which indicate which U-mode MM register pair is used. The general-purpose registers are shifted six bits to the left. The sixteen high-order bits of the result are the virtual address for U-mode reads or writes. This virtual address is used with the MFPI/MTPI instructions, with a special case if the zero bit is set: this indicates a byte address not on a word boundary. Unfortunately, the MFPI/MTPI instructions only transfer from/to word boundaries. To MFPI this first byte, the function MFPIs a word from the virtual address whose bit zero is cleared, then returns the high-order byte of the word fetched. To MTPI this first byte, the function MFPIs a word from the virtual address whose bit zero is cleared, then places the first byte in the high-order byte of the word fetched, finally it MTPIs the word back to the same virtual address. When all transfers are completed, the old values of the memory management registers are restored, and then the old value of the PSW is restored.

3.2.2 Instruction Backup. In order to increase a user's stack space, the UNIX operating system must be able to restart a user's instruction. To restart an instruction, all addressing-mode side effects on general-purpose registers must be undone. The addressing modes that have such side effects are auto-increment/decrement and explicit references through the program counter. The algorithm used to correct the general-purpose registers starts by fetching the instruction to be restarted and deciding upon the number and type of its addressing modes. The number and type of addressing modes are calculated by decoding bits 15-12 for all instructions, bits 11-9 for instructions with bits 15-12 equal to 0000, 1000, or 1111 (binary), and bits 8-6 for instructions with bits 15-9 equal to 1111000 (binary). The possible side effects on the general-purpose registers are calculated for each addressing mode, assuming a MM trap did not occur. A MM trap will cause instructions to be partially executed, which means that not *all* the side effects necessarily occur. Thus, it must be determined which addressing mode caused the MM trap in order to determine which side effects must be undone. If the instruction has one addressing mode affecting a general-purpose register, then that is the addressing mode that caused the fault. The general-purpose register is corrected and the routine exits. If the instruction has two addressing modes affecting general-purpose registers, the source and the destination addressing must be checked to determine which one caused the fault. If the source addressing mode

4. A *core click* is defined as 64 bytes of memory.

5. Thereby allowing the reading and writing of 8K bytes.

caused the fault, only the source general-purpose register is corrected. If the destination addressing mode caused the fault, both general-purpose registers are corrected. This algorithm is not capable of correcting the general-purpose registers for instructions using the same general-purpose register for both source and destination addressing modes with side effects. Fortunately, the C compiler does not generate instructions using this combination of addressing modes.

3.2.3 Program-Interrupt-Request Register. The UNIX operating system requires the ability to detect when the CPU is running at a priority level equal to or lower than a level determined by the program-interrupt-request (PIR) register. To exactly simulate this register, the CPU priority should be examined before each change in priority level. This is expensive in terms of CPU time and, fortunately, unnecessary for UNIX; it is sufficient to check the priority level before each return from an interrupt. Just before the return-from-interrupt instruction is executed, the simulated PIR register is examined. If it is zero, the normal return from an interrupt sequence is followed. Otherwise, a register is counted down from 7, as the high-order byte of the simulated PIR register is shifted left. When a one is shifted out of the high-order byte of the PIR, the count-down register contains the priority level that should be used to compare against. The register is shifted left, moved to the low-order byte of the PIR, shifted another 4 bits left, and *or*'ed to the low-order byte of the PIR. The setting up of this byte is necessary to correctly simulate the PIR register of the PDP-11/70. The register now contains the desired priority level. Bits 7-5 of this register are compared to bits 7-5 of the PSW that was saved on the kernel stack when the interrupt occurred. If the saved PSW is greater then the desired priority level, the normal return from an interrupt sequence is followed. Otherwise, the contents of the program-interrupt-request vector (locations 0242 and 0240 octal) are pushed on the kernel stack and a return-from-interrupt instruction is executed to simulate the the PIR interrupt.

3.2.4 Set-Priority-Level Instruction. The UNIX operating system must be able to change CPU priority levels upon entering and exiting critical sections of code. To change priority levels, the PDP-11/34 must use a combination of bit-set and bit-clear instructions on the PSW. To change priority levels, the PSW must be brought to the high-priority level by bit-setting the PSW with a 0340 (octal) and then dropped down to the desired priority level by bit-clearing the unwanted priority bits. Changing to a high-priority level ensures that interrupts of a lower-priority level are not granted until the proper time. The only two exceptions are changing to priority-level 7, which is done by bit-setting the PSW with a 0340 (octal), and changing to priority-level 0, which is done by bit-clearing the PSW with a 0340 (octal).

4. INCREASING EFFECTIVE KERNEL SPACE

After the machine-language assist functions were written, the UNIX operating system ran on the PDP-11/34. It had enough room for an RL01 disk driver, a DZ11 terminal multiplexer, 30 processes, 9 system buffers, and 70 inodes. That PDP-11/34 UNIX operating system utilized less than 64K bytes of memory. However, this did not leave room for more device drivers, other desired kernel functions, or growth of system tables. Because the virtual-address space is limited by the PDP-11/34's MM hardware, only the effective-address space of the kernel may be increased. This section discusses the possible algorithms to increase the effective-address space and their interaction with the UNIX operating system.

4.1 Buffers

The single largest resource within the UNIX operating system is dedicated to the I/O buffer pool. Each entry consists of a buffer header of 26 bytes and an actual buffer of 512 bytes. Because the UNIX operating system does not always require direct addressability of its system buffers, the buffers may be moved out of kernel-address space. There are two possible algorithms for moving the buffers out of kernel-address space.

The first algorithm changes a K-mode MM register whenever addressability of a given buffer is required. This algorithm is fast. However, effective use of space requires the operating system to have 16 buffers, which fully utilized the address space of a MM register. When using 10 to 15 buffers, the amount of CPU time spent in searching the buffer pool is equal to the CPU time spent in re-doing the I/O. Therefore, this algorithm is not well suited for the small number of buffers usually found in the PDP-11/34 operating system. The author chose not to implement this algorithm, but to implement the following algorithm.

The second algorithm does not require a kernel MM register. It copies the contents of the buffers outside the kernel-address space to the kernel, using the machine-language assist functions. This algorithm is slower, but better suited to the number of buffers in the PDP-11/34. The impact of this algorithm on the PDP-11/34 operating system required the author to change buffer content references to function calls that copy bytes, integers, longs, and arbitrary numbers of bytes between physical addresses and virtual addresses, and place a copy of the current inode in the user block. For the sake of efficiency, a small number of kernel-addressable buffers is also maintained. These buffers are used as in-core copies of super-blocks and by some I/O devices.

4.2 User Block

The UNIX operating system controls the execution of a user process by keeping information about the state of the process in a structure called a user block. The PDP-11/70 operating system uses a *windowing* algorithm to address the user block. The *windowing* algorithm requires changing a kernel MM register to map a user block into its address space. Thus, the user block (which resides in memory locations preceding the corresponding process) is addressed as part of the operating system. The user block occupies 1K bytes of the 8K bytes available for mapping by a MM register. This windowing algorithm allows the operating system to quickly exchange user blocks by modifying a MM register. In expanding the effective-address space for the PDP-11/34 operating system, the author chose to use a slower algorithm that exchanges user blocks by copying them between kernel-address space and the locations preceding the user process. The advantage of this approach is that the MM register used by the PDP-11/70 version to address the 1K byte user block is now used to map 8K bytes of kernel-address space. This results in a gain of 7K bytes of kernel-address space. This algorithm required changes to the routines that exchange user blocks. These routines involve saving the current state of a process and resuming the previous state. When a user process is saved, the kernel-addressable user block is saved in the memory locations preceding the process. When a user process is resumed, the kernel-addressable user block is restored from the memory locations preceding the process. For the sake of efficiency *setjmp* and *longjmp* routines have replaced *save* and *resume* routines where only non-local *goto*s were required.

4.3 Other Possibilities

The author investigated many other possibilities to increase the effective-address space of the PDP-11/34 UNIX operating system. Following is a list of ideas that were considered, with the reasons for their rejection:

1. Temporary removal of inactive inodes from kernel-address space. This would be a saving of 76 bytes per removed inode. The amount of code to implement this idea had a break-even point of about 15 inodes. In the PDP-11/34 system, there are rarely 15 inactive allocated inodes.

2. Export of read-only super-blocks. The amount of code to implement the moving of the read-only super-blocks out of kernel-address space outweighs the advantage of their removal due to the small number of read-only file systems.

3. Pruning of the operating system. Space can be recovered by removing infrequently used operating system functions. The error logger, *ptrace*, and profiling routines could be removed. The author feels this form of space saving should only be used as a last resort,

because the resulting system is no longer a true UNIX system.

4.4 Under Consideration

The author is currently investigating other possibilities to increase the effective-address space of the PDP-11/34 UNIX operating system. Following is a list of ideas that are being considered:

1. Implementing device drivers as user programs. Infrequently used device drivers may be written as user programs. This, coupled with a system call that gives the user addressability of the I/O page, could be an effective saver of space for such device drivers as magnetic tape and line printers.
2. Using segmentation overlay within the operating system. Infrequently used functions within the operating system could be placed outside of kernel-address space. When these functions are needed, the contents of the MM registers would be modified to place these segments in kernel-address space. This would be done (invisibly to both the operating system and to user programs) by modifying the loader and adding machine-language assist functions. The modifications would insert, at subroutine calls, code for invoking machine-language assist functions that would, in turn, modify appropriately the contents of the MM registers.

ACKNOWLEDGEMENT

I would like to thank Larry A. Wehr for advice that lead to the first version of UNIX for the PDP-11/34. I would like to especially thank Sharon Murrel and James Goodnow, II for being patient users of my many experimental operating systems.

REFERENCES

[1] 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.

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

UNIX Assembler Reference Manual

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

INTRODUCTION

This document describes the usage and input syntax of the UNIX† PDP-11 assembler *as*. The details of the PDP-11 are not described.

The input syntax of the UNIX assembler is generally similar to that of the DEC assembler PAL-11R, although its internal workings and output format are unrelated. It may be useful to read the publication DEC-11-ASDB-D, which describes PAL-11R, although naturally one must use care in assuming that its rules apply to *as*.

As is a rather ordinary assembler without macro capabilities. It produces an output file that contains relocation information and a complete symbol table; thus the output is acceptable to the UNIX link-editor *ld*, which may be used to combine the outputs of several assembler runs and to obtain object programs from libraries. The output format has been designed so that if a program contains no unresolved references to external symbols, it is executable without further processing.

1. USAGE

as is used as follows:

as [−**u**] [−**o** *output*] *file*$_1$ ...

If the optional "−u" argument is given, all undefined symbols in the current assembly will be made undefined-external. See the **.globl** directive below.

The other arguments name files which are concatenated and assembled. Thus programs may be written in several pieces and assembled together.

The output of the assembler is by default placed on the file *a.out* in the current directory; the "−o" flag causes the output to be placed on the named file. If there were no unresolved external references, and no errors detected, the output file is marked executable; otherwise, if it is produced at all, it is made non-executable.

2. LEXICAL CONVENTIONS

Assembler tokens include identifiers (alternatively, "symbols" or "names"), temporary symbols, constants, and operators.

2.1 Identifiers

An identifier consists of a sequence of alphanumeric characters (including period ".", underscore "_", and tilde "~" as alphanumeric) of which the first may not be numeric. Only the first eight characters are significant. When a name begins with a tilde, the tilde is discarded and that occurrence of the identifier generates a unique entry in the symbol table which can match no other occurrence of the identifier. This feature is used by the C compiler to place

† UNIX is a trademark of Bell Laboratories.

names of local variables in the output symbol table without having to worry about making them unique.

2.2 Temporary Symbols

A temporary symbol consists of a digit followed by "f" or "b". Temporary symbols are discussed fully in §5.1.

2.3 Constants

An octal constant consists of a sequence of digits; "8" and "9" are taken to have octal value 10 and 11. The constant is truncated to 16 bits and interpreted in two's complement notation.

A decimal constant consists of a sequence of digits terminated by a decimal point ".". The magnitude of the constant should be representable in 15 bits; i.e., be less than 32,768.

A single-character constant consists of a single quote "'" followed by an ASCII character not a new-line. Certain dual-character escape sequences are acceptable in place of the ASCII character to represent new-line and other non-graphics (see *String statements*, §5.5). The constant's value has the code for the given character in the least significant byte of the word and is null-padded on the left.

A double-character constant consists of a double quote '"' followed by a pair of ASCII characters not including new-line. Certain dual-character escape sequences are acceptable in place of either of the ASCII characters to represent new-line and other non-graphics (see *String statements*, §5.5). The constant's value has the code for the first given character in the least significant byte and that for the second character in the most significant byte.

2.4 Operators

There are several single- and double-character operators; see §6.

2.5 Blanks

Blank and tab characters may be interspersed freely between tokens, but may not be used within tokens (except character constants). A blank or tab is required to separate adjacent identifiers or constants not otherwise separated.

2.6 Comments

The character "/" introduces a comment, which extends through the end of the line on which it appears. Comments are ignored by the assembler.

3. SEGMENTS

Assembled code and data fall into three segments: the text segment, the data segment, and the bss segment. The text segment is the one in which the assembler begins, and it is the one into which instructions are typically placed. The UNIX system will, if desired, enforce the purity of the text segment of programs by trapping write operations into it. Object programs produced by the assembler must be processed by the link-editor *ld* (using its "−n" flag) if the text segment is to be write-protected. A single copy of the text segment is shared among all processes executing such a program.

The data segment is available for placing data or instructions which will be modified during execution. Anything which may go in the text segment may be put into the data segment. In programs with write-protected, sharable text segments, data segment contains the initialized but variable parts of a program. If the text segment is not pure, the data segment begins immediately after the text segment; if the text segment is pure, the data segment begins at the lowest 8K byte boundary after the text segment.

The bss segment may not contain any explicitly initialized code or data. The length of the bss segment (like that of text or data) is determined by the high-water mark of the location counter within it. The bss segment is actually an extension of the data segment and begins immediately after it. At the start of execution of a program, the bss segment is set to 0. Typically the bss segment is set up by statements exemplified by:

```
lab: . = .+10
```

The advantage in using the bss segment for storage that starts off empty is that the initialization information need not be stored in the output file. See also *Location counter* and *Assignment statements* below.

4. THE LOCATION COUNTER

One special symbol, ".", is the location counter. Its value at any time is the offset within the appropriate segment of the start of the statement in which it appears. The location counter may be assigned to, with the restriction that the current segment may not change; furthermore, the value of "." may not decrease. If the effect of the assignment is to increase the value of ".", the required number of null bytes are generated (but see *Segments* above).

5. STATEMENTS

A source program is composed of a sequence of *statements*. Statements are separated either by new-lines or by semicolons. There are five kinds of statements: null statements, expression statements, assignment statements, string statements, and keyword statements.

Any kind of statement may be preceded by one or more labels.

5.1 Labels

There are two kinds of label: name labels and numeric labels. A name label consists of a name followed by a colon (:). The effect of a name label is to assign the current value and type of the location counter "." to the name. An error is indicated in pass 1 if the name is already defined; an error is indicated in pass 2 if the "." value assigned changes the definition of the label.

A numeric label consists of a digit *0* to *9* followed by a colon (:). Such a label serves to define temporary symbols of the form "*n*b" and "*n*f", where *n* is the digit of the label. As in the case of name labels, a numeric label assigns the current value and type of "." to the temporary symbol. However, several numeric labels with the same digit may be used within the same assembly. References of the form "*n*f" refer to the first numeric label "*n*:" *f*orward from the reference; "*n*b" symbols refer to the first "*n*:" label *b*ackward from the reference. This sort of temporary label was introduced by Knuth [*The Art of Computer Programming, Vol. I: Fundamental Algorithms*]. Such labels tend to conserve both the symbol table space of the assembler and the inventive powers of the programmer.

5.2 Null Statements

A null statement is an empty statement (which may, however, have labels). A null statement is ignored by the assembler. Common examples of null statements are empty lines or lines containing only a label.

5.3 Expression Statements

An expression statement consists of an arithmetic expression not beginning with a keyword. The assembler computes its (16-bit) value and places it in the output stream, together with the appropriate relocation bits.

5.4 Assignment Statements

An assignment statement consists of an identifier, an equals sign (=), and an expression. The value and type of the expression are assigned to the identifier. It is not required that the type or value be the same in pass 2 as in pass 1, nor is it an error to redefine any symbol by assignment.

Any external attribute of the expression is lost across an assignment. This means that it is not possible to declare a global symbol by assigning to it, and that it is impossible to define a symbol to be offset from a non-locally defined global symbol.

As mentioned, it is permissible to assign to the location counter ".". It is required, however, that the type of the expression assigned be of the same type as ".", and it is forbidden to decrease the value of ".". In practice, the most common assignment to "." has the form ". = . + *n*" for some number *n;* this has the effect of generating *n* null bytes.

5.5 String Statements

A string statement generates a sequence of bytes containing ASCII characters. A string statement consists of a left string quote "<" followed by a sequence of ASCII characters not including new-line, followed by a right string quote ">". Any of the ASCII characters may be replaced by a two-character escape sequence to represent certain non-graphic characters, as follows:

\n	NL	(012)
\s	SP	(040)
\t	HT	(011)
\e	EOT	(004)
\0	NUL	(000)
\r	CR	(015)
\a	ACK	(006)
\p	PFX	(033)
\\	\	
\>	>	

The last two are included so that the escape character and the right string quote may be represented. The same escape sequences may also be used within single- and double-character constants (see §2.3 above).

5.6 Keyword Statements

Keyword statements are numerically the most common type, since most machine instructions are of this sort. A keyword statement begins with one of the many predefined keywords of the assembler; the syntax of the remainder depends on the keyword. All the keywords are listed below with the syntax they require.

6. EXPRESSIONS

An expression is a sequence of symbols representing a value. Its constituents are identifiers, constants, temporary symbols, operators, and brackets. Each expression has a type.

All operators in expressions are fundamentally binary in nature; if an operand is missing on the left, a 0 of absolute type is assumed. Arithmetic is two's complement and has 16 bits of precision. All operators have equal precedence, and expressions are evaluated strictly left to right except for the effect of brackets.

6.1 Expression Operators

The operators are:

(blank) when there is no operator between operands, the effect is exactly the same as if a "+" had appeared.

\+ addition

\- subtraction

* multiplication

\/ division (note that plain " / " starts a comment)

& bitwise **and**

| bitwise **or**

\> logical right shift

\< logical left shift

% modulo

! *a*!*b* is *a* **or** (**not** *b*); i.e., the **or** of the first operand and the one's complement of the second; most common use is as a unary.

^ result has the value of first operand and the type of the second; most often used to define new machine instructions with syntax identical to existing instructions.

Expressions may be grouped by use of square brackets " [ ] ". (Round parentheses are reserved for address modes.)

6.2 Types

The assembler deals with a number of types of expressions. Most types are attached to keywords and used to select the routine which treats that keyword. The types likely to be met explicitly are:

undefined

Upon first encounter, each symbol is undefined. It may become undefined if it is assigned an undefined expression. It is an error to attempt to assemble an undefined expression in pass 2; in pass 1, it is not (except that certain keywords require operands which are not undefined).

undefined external

A symbol which is declared **.globl** but not defined in the current assembly is an undefined external. If such a symbol is declared, the link editor *ld* must be used to load the assembler's output with another routine that defines the undefined reference.

absolute An absolute symbol is defined ultimately from a constant. Its value is unaffected by any possible future applications of the link-editor to the output file.

text The value of a text symbol is measured with respect to the beginning of the text segment of the program. If the assembler output is link-edited, its text symbols may change in value since the program need not be the first in the link editor's output. Most text symbols are defined by appearing as labels. At the start of an assembly, the value of " . " is text 0.

data The value of a data symbol is measured with respect to the origin of the data segment of a program. Like text symbols, the value of a data symbol may change during a subsequent link-editor run since previously loaded programs may have data segments. After the first **.data** statement, the value of " . " is data 0.

bss The value of a bss symbol is measured from the beginning of the bss segment of a program. Like text and data symbols, the value of a bss symbol may change during a subsequent link-editor run, since previously loaded programs may have bss segments. After the first **.bss** statement, the value of " . " is bss 0.

external absolute, text, data, or bss

Symbols declared **.globl** but defined within an assembly as absolute, text, data, or bss symbols may be used exactly as if they were not declared **.globl**; however, their value and type are available to the link editor so that the program may be loaded with others that reference these symbols.

register The symbols

r0 ... r5 fr0 ... fr5 sp pc

are predefined as register symbols. Either they or symbols defined from them must be used to refer to the 6 general-purpose, 6 floating-point, and the 2 special-purpose machine registers. The behavior of the floating register names is identical to that of the corresponding general register names; the former are provided as a mnemonic aid.

other types

Each keyword known to the assembler has a type which is used to select the routine which processes the associated keyword statement. The behavior of such symbols when not used as keywords is the same as if they were absolute.

6.3 Type Propagation in Expressions

When operands are combined by expression operators, the result has a type which depends on the types of the operands and on the operator. The rules involved are complex to state but were intended to be sensible and predictable. For purposes of expression evaluation the important types are:

undefined
absolute
text
data
bss
undefined external
other

The combination rules are then: If one of the operands is undefined, the result is undefined. If both operands are absolute, the result is absolute. If an absolute is combined with one of the "other types" mentioned above, or with a register expression, the result has the register or other type. As a consequence, one can refer to r3 as "r0+3". If two operands of "other type" are combined, the result has the numerically larger type An "other type" combined with an explicitly discussed type other than absolute acts like an absolute.

Further rules applying to particular operators are:

\+ If one operand is text-, data-, or bss-segment relocatable, or is an undefined external, the result has the postulated type and the other operand must be absolute.

\- If the first operand is a relocatable text-, data-, or bss-segment symbol, the second operand may be absolute (in which case the result has the type of the first operand); or the second operand may have the same type as the first (in which case the result is absolute). If the first operand is external undefined, the second must be absolute. All other combinations are illegal.

* This operator follows no other rule than that the result has the value of the first operand and the type of the second.

others It is illegal to apply these operators to any but absolute symbols.

7. PSEUDO-OPERATIONS

The keywords listed below introduce statements that generate data in unusual forms or influence the later operations of the assembler. The metanotation

[stuff] ...

means that 0 or more instances of the given stuff may appear. Also, boldface tokens are literals, italic words are substitutable.

7.1 .byte *expression* [, *expression*] ...

The *expressions* in the comma-separated list are truncated to 8 bits and assembled in successive bytes. The expressions must be absolute. This statement and the string statement above are the only ones that assemble data one byte at at time.

7.2 .even

If the location counter "." is odd, it is advanced by one so the next statement will be assembled at a word boundary.

7.3 .if *expression*

The *expression* must be absolute and defined in pass 1. If its value is nonzero, the **.if** is ignored; if zero, the statements between the **.if** and the matching **.endif** (below) are ignored. **.if** may be nested. The effect of **.if** cannot extend beyond the end of the input file in which it appears. (The statements are not totally ignored, in the following sense: **.if**s and **.endif**s are scanned for, and moreover all names are entered in the symbol table. Thus names occurring only inside an **.if** will show up as undefined if the symbol table is listed.)

7.4 .endif

This statement marks the end of a conditionally-assembled section of code. See **.if** above.

7.5 .globl *name* [, *name*] ...

This statement makes the *names* external. If they are otherwise defined (by assignment or appearance as a label) they act within the assembly exactly as if the **.globl** statement were not given; however, the link editor *ld* may be used to combine this routine with other routines that refer these symbols.

Conversely, if the given symbols are not defined within the current assembly, the link editor can combine the output of this assembly with that of others which define the symbols. As discussed in §1, it is possible to force the assembler to make all otherwise undefined symbols external.

7.6 .text
7.7 .data
7.8 .bss

These three pseudo-operations cause the assembler to begin assembling into the text, data, or bss segment respectively. Assembly starts in the text segment. It is forbidden to assemble any code or data into the bss segment, but symbols may be defined and "." moved about by assignment.

7.9 .comm *name* , *expression*

Provided the *name* is not defined elsewhere, this statement is equivalent to

```
.globl name
name = expression ^ name
```

That is, the type of *name* is "undefined external", and its value is *expression*. In fact the *name* behaves in the current assembly just like an undefined external. However, the link-editor *ld* has been special-cased so that all external symbols which are not otherwise defined, and which have a non-zero value, are defined to lie in the bss segment, and enough space is left after the symbol to hold *expression* bytes. All symbols which become defined in this way are located before all the explicitly defined bss-segment locations.

8. MACHINE INSTRUCTIONS

Because of the rather complicated instruction and addressing structure of the PDP-11, the syntax of machine instruction statements is varied. Although the following sections give the syntax in detail, the machine handbooks should be consulted on the semantics.

8.1 Sources and Destinations

The syntax of general source and destination addresses is the same. Each must have one of the following forms, where *reg* is a register symbol, and *expr* is any sort of expression:

syntax	words	mode
reg	0	00+*reg*
(*reg*) +	0	20+*reg*
− (*reg*)	0	40+*reg*
expr (*reg*)	1	60+*reg*
(*reg*)	0	10+*reg*
* *reg*	0	10+*reg*
* (*reg*) +	0	30+*reg*
* − (*reg*)	0	50+*reg*
* (*reg*)	1	70+*reg*
* *expr* (*reg*)	1	70+*reg*
expr	1	67
$ *expr*	1	27
* *expr*	1	77
* $ *expr*	1	37

The *words* column gives the number of address words generated; the *mode* column gives the octal address-mode number. The syntax of the address forms is identical to that in DEC assemblers, except that "*" has been substituted for "@" and "$" for "#"; the UNIX typing conventions make "@" and "#" rather inconvenient.

Notice that mode "*reg" is identical to "(reg)"; that "*(reg)" generates an index word (namely, 0); and that addresses consisting of an unadorned expression are assembled as pc-relative references independent of the type of the expression. To force a non-relative reference, the form "*$expr" can be used, but notice that further indirection is impossible.

8.3 Simple Machine Instructions

The following instructions are defined as absolute symbols:

clc	**sec**
clv	**sev**
clz	**sez**
cln	**sen**

They therefore require no special syntax. The PDP-11 hardware allows more than one of the "clear" class, or alternatively more than one of the "set" class to be **or**-ed together; this may be expressed as follows:

```
clc | clv
```

8.4 Branch

The following instructions take an expression as operand. The expression must lie in the same segment as the reference, cannot be undefined-external, and its value cannot differ from the current location of "." by more than 254 bytes:

br	**blos**	
bne	**bvc**	
beq	**bvs**	
bge	**bhis**	
blt	**bec**	(= **bcc**)
bgt	**bcc**	
ble	**blo**	
bpl	**bcs**	
bmi	**bes**	(= **bcs**)
bhi		

bes ("branch on error set") and **bec** ("branch on error clear") are intended to test the error bit returned by system calls (which is the c-bit).

8.5 Extended Branch Instructions

The following symbols are followed by an expression representing an address in the same segment as "."; if the target address is close enough, a branch-type instruction is generated; if the address is too far away, a **jmp** will be used:

jbr	**jlos**
jne	**jvc**
jeq	**jvs**
jge	**jhis**
jlt	**jec**
jgt	**jcc**
jle	**jlo**
jpl	**jcs**
jmi	**jes**
jhi	

jbr turns into a plain **jmp** if its target is too remote; the others (whose names are constructed by replacing the "b" in the branch instruction's name by "j") turn into the converse branch over a **jmp** to the target address.

8.6 Single Operand Instructions

The following symbols are names of single-operand machine instructions. The form of address expected is discussed in §8.1 above:

clr	**sbcb**
clrb	**ror**
com	**rorb**
comb	**rol**
inc	**rolb**
incb	**asr**
dec	**asrb**
decb	**asl**
neg	**aslb**
negb	**jmp**
adc	**swab**
adcb	**tst**
sbc	**tstb**

8.7 Double Operand Instructions

The following instructions take a general source and destination (§8.1), separated by a comma, as operands:

mov	**bic**
movb	**bicb**
cmp	**bis**
cmpb	**bisb**
bit	**add**
bitb	**sub**

8.8 Miscellaneous Instructions

The following instructions have a more specialized syntax. Here *reg* is a register name, *src* and *dst* a general source or destination (§8.1), and *expr* is an expression:

jsr	*reg,dst*	
rts	*reg*	
sys	*expr*	
ash	*src*, *reg*	(or **als**)
ashc	*src*, *reg*	(or **alsc**)
mul	*src*, *reg*	(or **mpy**)
div	*src*, *reg*	(or **dvd**)
xor	*reg*, *dst*	
sxt	*dst*	
mark	*expr*	
sob	*reg*, *expr*	

sys is another name for the **trap** instruction. It is used to code system calls. Its operand is required to be expressible in 6 bits. The expression in **mark** must be expressible in 6 bits, and the expression in **sob** must be in the same segment as ".", must not be external-undefined, must be less than ".", and must be within 510 bytes of ".".

8.9 Floating-Point Unit Instructions

The following floating-point operations are defined, with syntax as indicated:

cfcc		
setf		
setd		
seti		
setl		
clrf	*fdst*	
negf	*fdst*	
absf	*fdst*	
tstf	*fsrc*	
movf	*fsrc,freg*	(= ldf)
movf	*freg,fdst*	(= stf)
movif	*src,freg*	(= ldcif)
movfi	*freg, dst*	(= stcfi)
movof	*fsrc,freg*	(= ldcdf)
movfo	*freg,fdst*	(= stcfd)
movie	*src,freg*	(= ldexp)
movei	*freg, dst*	(= stexp)
addf	*fsrc,freg*	
subf	*fsrc,freg*	
mulf	*fsrc,freg*	
divf	*fsrc,freg*	
cmpf	*fsrc,freg*	
modf	*fsrc,freg*	
ldfps	*src*	
stfps	*dst*	
stst	*dst*	

fsrc, *fdst*, and *freg* mean floating-point source, destination, and register respectively. Their syntax is identical to that for their non-floating counterparts, but note that only floating registers 0-3 can be a *freg*.

The names of several of the operations have been changed to bring out an analogy with certain fixed-point instructions. The only strange case is **movf**, which turns into either **stf** or **ldf** depending respectively on whether its first operand is or is not a register. Warning: **ldf** sets the floating condition codes, **stf** does not.

9. OTHER SYMBOLS

9.1 ..

The symbol "." is the *relocation counter*. Just before each assembled word is placed in the output stream, the current value of this symbol is added to the word if the word refers to a text, data or bss segment location. If the output word is a pc-relative address word that refers to an absolute location, the value of "." is subtracted. Thus the value of "." can be taken to mean the starting memory location of the program. The initial value of "." is 0.

The value of "." may be changed by assignment. Such a course of action is sometimes necessary, but the consequences should be carefully thought out. It is particularly ticklish to change "." midway in an assembly or to do so in a program which will be treated by the loader, which has its own notions of ".".

9.2 System Calls

System call names are not predefined. They may be found in the file */usr/include/sys.s*.

10. DIAGNOSTICS

When an input file cannot be read, its name followed by a question mark is typed and assembly ceases. When syntactic or semantic errors occur, a single-character diagnostic is typed out together with the line number and the file name in which it occurred. Errors in pass 1 cause cancellation of pass 2. The possible errors are:

)	parentheses error
]	parentheses error
>	string not terminated properly
*	indirection (*) used illegally
.	illegal assignment to "."
A	error in address
B	branch address is odd or too remote
E	error in expression
F	error in local ("f" or "b") type symbol
G	garbage (unknown) character
I	end of file inside an **.if**
M	multiply defined symbol as label
O	word quantity assembled at odd address
P	phase error— "." different in pass 1 and 2
R	relocation error
U	undefined symbol
X	syntax error

A Tour Through the Portable C Compiler

S. C. Johnson

Bell Laboratories
Murray Hill, New Jersey 07974

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 requires 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

$$\text{ARY} + (\text{PTR}<<2) + (\text{FTN}<<4) + \text{DOUBLE}$$

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 functions 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 non-reentrant. 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

$$a += b$$

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, resulting in 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 Address form of the Left 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

$$x\,{+}{+}$$

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,* Bell Laboratories (1978).

[3] A. Snyder. *A Portable Compiler for the Language C,* Master's Thesis, M.I.T., Cambridge, MA (1974).

[4] S. C. Johnson. "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages,* pp. 97-104 (Jan. 1978).

[5] M. E. Lesk, S. C. Johnson, and D. M. Ritchie. *The C Language Calling Sequence,* Bell Laboratories (1977).

[6] S. C. Johnson. *YACC—Yet Another Compiler-Compiler,* Bell Laboratories (July 1975).

[7] A. V. Aho and S. C. Johnson. "Optimal Code Generation for Expression Trees," *J. Assoc. Comp. Mach.* **23**(3):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.* **17**(4):715-728 (Oct. 1970). Reprinted in *Compiler Techniques,* ed. B. W. Pollack, pp. 229-247, Auerbach, Princeton, New Jersey (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 (Jan. 1977).

January 1981

A Tour Through the UNIX C Compiler

D. M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

The Intermediate Language

Communication between the two phases of the UNIX† C 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

† UNIX is a trademark of Bell Laboratories.

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 uninitialized 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 ease 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 appropriate *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 compatible 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 addressable (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:

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 addressable 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 addressable; 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 addressable 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 addressable; 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 addressable. 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)*

```
%c*,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 labeled and referred to later. It turns out, for example, that rather large portions of the the *effiab* 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 optimization. Many cases that 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.

January 1981

On the Security of UNIX

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

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

† UNIX is a trademark of Bell Laboratories.

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 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.

January 1981

Password Security—A Case History

Robert Morris

Ken Thompson

Bell Laboratories
Murray Hill, New Jersey 07974

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.

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

† UNIX is a trademark of Bell Laboratories.

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.

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 typically 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. *CACM* **17**(7):365-75 (July 1975).

[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.

January 1981