PLEXUS

Plexus Sys3 UNIX Programmer's Manual -- vol 2B

98-05037.5 Rev A September 24, 1984

PLEXUS

Sys3 UNIX vol 2B Programmer's Manual

Plexus Sys3 UNIX Programmer's Manual -- vol 2B

98-05037.5 Rev A September 24, 1984

PLEXUS COMPUTERS, INC.

3833 North First St.

San Jose, CA 95134

408/943-9433

Printed in the United States of America

Programmer's Manual for UNIX* System III

Volume 2 - Supplementary Documents

January 1983

This volume contains documents that supplement the information contained in the *Plexus Sys3 UNIX Programmer's Manual - vol 1*. The documents are grouped roughly into the areas of General Works, Basics, Document Preparation Tools, Programming and Language Tools, and System Administration and Maintenance Tools. Further general information may be found in the July - August 1978 special issue of "The Bell System Technical Journal" on the UNIX Time Sharing System.

These documents contain occasional localisms, typically references to other operating systems like GCOS and IBM. In all cases, such references may be safely ignored by users of UNIX systems.

---

* UNIX is a trademark of AT&T Bell Laboratories.

Plexus Sys3 UNIX Programmer's Manual -- vol 2B

PREFACE

This manual contains a collection of documents that describe specific aspects of the UNIX* operating system. These include descriptions of programming, language, administrative and maintenance tools.

Additional documents describing the operating system, document preparation tools and programming and language tools are collected in the *Plexus Sys3 UNIX Programmer's Manual -- vol 2A* (Plexus publication number 98-05036).

Both these volumes (2A and 2B) should be used as supplementary documents for the *Plexus Sys3 UNIX Programmer's Manual -- vol 1A* (Plexus publication number 98-05045) and *Plexus Sys3 UNIX Programmer's Manual -- vol 1B* (Plexus publication number 98-05046), the basic reference manual for the operating system.

Comments

Please address all comments concerning this manual to:

Plexus Computers, Inc.
Technical Publications Dept.
3833 North First St.
San Jose, CA 95134
408/943-9433

Revision History

The second edition (#98-05037.2) contained new front matter.

The third edition (#98-05037.3) contained a new VPM document.

For the fourth edition (#98-05037.4), several documents were re-typeset.

This edition (#98-05037.5) re-typesets several other documents and includes an updated version of the VPM documents.

---

* UNIX is a trademark of AT&T Bell Laboratories. Plexus is licensed to distribute UNIX under the authority of AT&T.

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.

November 12, 1978

†UNIX is a Trademark of Bell Laboratories.

BC — An Arbitrary Precision Desk-Calculator Language

Lorinda Cherry

Robert Morris

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

BC is a language and a compiler for doing arbitrary precision arithmetic on the UNIX† time-sharing system [1]. The compiler was written to make conveniently available a collection of routines (called DC [5]) which are capable of doing arithmetic on integers of arbitrary size. The compiler is by no means intended to provide a complete programming language. It is a minimal language facility.

There is a scaling provision that permits the use of decimal point notation. Provision is made for input and output in bases other than decimal. Numbers can be converted from decimal to octal by simply setting the output base to equal 8.

The actual limit on the number of digits that can be handled depends on the amount of storage available on the machine. Manipulation of numbers with many hundreds of digits is possible even on the smallest versions of UNIX.

The syntax of BC has been deliberately selected to agree substantially with the C language [2]. Those who are familiar with C will find few surprises in this language.

Simple Computations with Integers

The simplest kind of statement is an arithmetic expression on a line by itself. For instance, if you type in the line:

```
142857 + 285714
```

the program responds immediately with the line

```
428571
```

The operators −, *, /, %, and ^ can also be used; they indicate subtraction, multiplication, division, remaindering, and exponentiation, respectively. Division of integers produces an integer result truncated toward zero. Division by zero produces an error comment.

Any term in an expression may be prefixed by a minus sign to indicate that it is to be negated (the 'unary' minus sign). The expression

```
7+-3
```

is interpreted to mean that −3 is to be added to 7.

More complex expressions with several operators and with parentheses are interpreted just as in Fortran, with ^ having the greatest binding power, then * and % and /, and finally + and −. Contents of parentheses are evaluated before material outside the parentheses. Exponentiations are performed from right to left and the other operators from left to right. The two expressions

---

†UNIX is a Trademark of Bell Laboratories.

a^b^c and a^(b^c)

are equivalent, as are the two expressions

a*b*c and (a*b)*c

BC shares with Fortran and C the undesirable convention that

a/b*c is equivalent to (a/b)*c

Internal storage registers to hold numbers have single lower-case letter names. The value of an expression can be assigned to a register in the usual way. The statement

```
x = x + 3
```

has the effect of increasing by three the value of the contents of the register named x. When, as in this case, the outermost operator is an =, the assignment is performed but the result is not printed. Only 26 of these named storage registers are available.

There is a built-in square root function whose result is truncated to an integer (but see scaling below). The lines

```
x = sqrt(191)
x
```

produce the printed result

```
13
```

Bases

There are special internal quantities, called 'ibase' and 'obase'. The contents of 'ibase', initially set to 10, determines the base used for interpreting numbers read in. For example, the lines

```
ibase = 8
11
```

will produce the output line

```
9
```

and you are all set up to do octal to decimal conversions. Beware, however of trying to change the input base back to decimal by typing

```
ibase = 10
```

Because the number 10 is interpreted as octal, this statement will have no effect. For those who deal in hexadecimal notation, the characters A−F are permitted in numbers (no matter what base is in effect) and are interpreted as digits having values 10−15 respectively. The statement

```
ibase = A
```

will change you back to decimal input base no matter what the current input base is. Negative and large positive input bases are permitted but useless. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 and greater than 16.

The contents of 'obase', initially set to 10, are used as the base for output numbers. The lines

```
obase = 16
1000
```

will produce the output line

```
3E8
```

which is to be interpreted as a 3-digit hexadecimal number. Very large output bases are permitted, and they are sometimes useful. For example, large numbers can be output in groups of five digits by setting 'obase' to 100000. Strange (i.e. 1, 0, or negative) output bases are handled appropriately.

Very large numbers are split across lines with 70 characters per line. Lines which are continued end with \. Decimal output conversion is practically instantaneous, but output of very large numbers (i.e., more than 100 digits) with other bases is rather slow. Non-decimal output conversion of a one hundred digit number takes about three seconds.

It is best to remember that 'ibase' and 'obase' have no effect whatever on the course of internal computation or on the evaluation of expressions, but only affect input and output conversion, respectively.

Scaling

A third special internal quantity called 'scale' is used to determine the scale of calculated quantities. Numbers may have up to 99 decimal digits after the decimal point. This fractional part is retained in further computations. We refer to the number of digits after the decimal point of a number as its scale.

When two scaled numbers are combined by means of one of the arithmetic operations, the result has a scale determined by the following rules. For addition and subtraction, the scale of the result is the larger of the scales of the two operands. In this case, there is never any truncation of the result. For multiplications, the scale of the result is never less than the maximum of the two scales of the operands, never more than the sum of the scales of the operands and, subject to those two restrictions, the scale of the result is set equal to the contents of the internal quantity 'scale'. The scale of a quotient is the contents of the internal quantity 'scale'. The scale of a remainder is the sum of the scales of the quotient and the divisor. The result of an exponentiation is scaled as if the implied multiplications were performed. An exponent must be an integer. The scale of a square root is set to the maximum of the scale of the argument and the contents of 'scale'.

All of the internal operations are actually carried out in terms of integers, with digits being discarded when necessary. In every case where digits are discarded, truncation and not rounding is performed.

The contents of 'scale' must be no greater than 99 and no less than 0. It is initially set to 0. In case you need more than 99 fraction digits, you may arrange your own scaling.

The internal quantities 'scale', 'ibase', and 'obase' can be used in expressions just like other variables. The line

```
scale = scale + 1
```

increases the value of 'scale' by one, and the line

```
scale
```

causes the current value of 'scale' to be printed.

The value of 'scale' retains its meaning as a number of decimal digits to be retained in internal computation even when 'ibase' or 'obase' are not equal to 10. The internal computations (which are still conducted in decimal, regardless of the bases) are performed to the specified number of decimal digits, never hexadecimal or octal or any other kind of digits.

Functions

The name of a function is a single lower-case letter. Function names are permitted to collide with simple variable names. Twenty-six different defined functions are permitted in addition to the twenty-six variable names. The line

```
define a(x){
```

begins the definition of a function with one argument. This line must be followed by one or more statements, which make up the body of the function, ending with a right brace }. Return of control from a function occurs when a return statement is executed or when the end of the function is reached. The return statement can take either of the two forms

```
return
return(x)
```

In the first case, the value of the function is 0, and in the second, the value of the expression in parentheses.

Variables used in the function can be declared as automatic by a statement of the form

```
auto x,y,z
```

There can be only one 'auto' statement in a function and it must be the first statement in the definition. These automatic variables are allocated space and initialized to zero on entry to the function and thrown away on return. The values of any variables with the same names outside the function are not disturbed. Functions may be called recursively and the automatic variables at each level of call are protected. The parameters named in a function definition are treated in the same way as the automatic variables of that function with the single exception that they are given a value on entry to the function. An example of a function definition is

```
define a(x,y){
        auto z
        z = x*y
        return(z)
}
```

The value of this function, when called, will be the product of its two arguments.

A function is called by the appearance of its name followed by a string of arguments enclosed in parentheses and separated by commas. The result is unpredictable if the wrong number of arguments is used.

Functions with no arguments are defined and called using parentheses with nothing between them: b().

If the function *a* above has been defined, then the line

```
a(7,3.14)
```

would cause the result 21.98 to be printed and the line

```
x = a(a(3,4),5)
```

would cause the value of x to become 60.

Subscripted Variables

A single lower-case letter variable name followed by an expression in brackets is called a subscripted variable (an array element). The variable name is called the array name and the expression in brackets is called the subscript. Only one-dimensional arrays are permitted. The names of arrays are permitted to collide with the names of simple variables and function names. Any fractional part of a subscript is discarded before use. Subscripts must be greater than or equal to zero and less than or equal to 2047.

Subscripted variables may be freely used in expressions, in function calls, and in return statements.

An array name may be used as an argument to a function, or may be declared as automatic in a function definition by the use of empty brackets:

```
f(a[])
define f(a[])
auto a[]
```

When an array name is so used, the whole contents of the array are copied for the use of the function, and thrown away on exit from the function. Array names which refer to whole arrays cannot be used in any other contexts.

Control Statements

The 'if', the 'while', and the 'for' statements may be used to alter the flow within programs or to cause iteration. The range of each of them is a statement or a compound statement consisting of a collection of statements enclosed in braces. They are written in the following way

```
if(relation) statement
while(relation) statement
for(expression1; relation; expression2) statement
```

or

```
if(relation) {statements}
while(relation) {statements}
for(expression1; relation; expression2) {statements}
```

A relation in one of the control statements is an expression of the form

```
x>y
```

where two expressions are related by one of the six relational operators <, >, <=, >=, ==, or !=. The relation == stands for 'equal to' and != stands for 'not equal to'. The meaning of the remaining relational operators is clear.

BEWARE of using = instead of == in a relational. Unfortunately, both of them are legal, so you will not get a diagnostic message, but = really will not do a comparison.

The 'if' statement causes execution of its range if and only if the relation is true. Then control passes to the next statement in sequence.

The 'while' statement causes execution of its range repeatedly as long as the relation is true. The relation is tested before each execution of its range and if the relation is false, control passes to the next statement beyond the range of the while.

The 'for' statement begins by executing 'expression1'. Then the relation is tested and, if true, the statements in the range of the 'for' are executed. Then 'expression2' is executed. The relation is tested, and so on. The typical use of the 'for' statement is for a controlled iteration, as in the statement

```
for(i=1; i<=10; i=i+1) i
```

which will print the integers from 1 to 10. Here are some examples of the use of the control statements.

```
define f(n){
auto i, x
x=1
for(i=1; i<=n; i=i+1) x=x*i
return(x)
}
```

The line

```
f(a)
```

will print *a* factorial if *a* is a positive integer. Here is the definition of a function which will compute values of the binomial coefficient (m and n are assumed to be positive integers).

```
define b(n,m){
auto x, j
x=1
for(j=1; j<=m; j=j+1) x=x*(n-j+1)/j
return(x)
}
```

The following function computes values of the exponential function by summing the appropriate series without regard for possible truncation errors:

```
scale = 20
define e(x){
        auto a, b, c, d, n
        a = 1
        b = 1
        c = 1
        d = 0
        n = 1
        while(1==1){
                a = a*x
                b = b*n
                c = c + a/b
                n = n + 1
                if(c==d) return(c)
                d = c
        }
}
```

Some Details

There are some language features that every user should know about even if he will not use them.

Normally statements are typed one to a line. It is also permissible to type several statements on a line separated by semicolons.

If an assignment statement is parenthesized, it then has a value and it can be used anywhere that an expression can. For example, the line

```
(x=y+17)
```

not only makes the indicated assignment, but also prints the resulting value.

Here is an example of a use of the value of an assignment statement even when it is not parenthesized.

```
x = a[i=i+1]
```

causes a value to be assigned to x and also increments i before it is used as a subscript.

The following constructs work in BC in exactly the same manner as they do in the C language. Consult the appendix or the C manuals [2] for their exact workings.

x=y=z is the same as	x=(y=z)
x =+ y	x = x+y
x =− y	x = x−y
x =* y	x = x*y
x =/ y	x = x/y
x =% y	x = x%y
x =^ y	x = x^y
x++	(x=x+1)−1
x−−	(x=x−1)+1
++x	x = x+1
−−x	x = x−1

Even if you don't intend to use the constructs, if you type one inadvertently, something correct but unexpected may happen.

WARNING! In some of these constructions, spaces are significant. There is a real difference between x=−y and x= −y. The first replaces x by x−y and the second by −y.

Three Important Things

1. To exit a BC program, type 'quit'.

2. There is a comment convention identical to that of C and of PL/I. Comments begin with '/*' and end with '*/'.

3. There is a library of math functions which may be obtained by typing at command level

```
bc −l
```

This command will load a set of library functions which, at the time of writing, consists of sine (named 's'), cosine ('c'), arctangent ('a'), natural logarithm ('l'), exponential ('e') and Bessel functions of integer order ('j(n,x)'). Doubtless more functions will be added in time. The library sets the scale to 20. You can reset it to something else if you like. The design of these mathematical library routines is discussed elsewhere [3].

If you type

```
bc file ...
```

BC will read and execute the named file or files before accepting commands from the keyboard. In this way, you may load your favorite programs and function definitions.

Acknowledgement

The compiler is written in YACC [4]; its original version was written by S. C. Johnson.

References

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

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

[3] R. Morris, *A Library of Reference Standard Mathematical Subroutines,* Bell Laboratories internal memorandum, 1975.

[4] S. C. Johnson, *YACC — Yet Another Compiler-Compiler.* Bell Laboratories Computing Science Technical Report #32, 1978.

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

Appendix

1. Notation

In the following pages syntactic categories are in *italics*; literals are in **bold**; material in brackets [] is optional.

2. Tokens

Tokens consist of keywords, identifiers, constants, operators, and separators. Token separators may be blanks, tabs or comments. Newline characters or semicolons separate statements.

2.1. Comments

Comments are introduced by the characters /* and terminated by */.

2.2. Identifiers

There are three kinds of identifiers – ordinary identifiers, array identifiers and function identifiers. All three types consist of single lower-case letters. Array identifiers are followed by square brackets, possibly enclosing an expression describing a subscript. Arrays are singly dimensioned and may contain up to 2048 elements. Indexing begins at zero so an array may be indexed from 0 to 2047. Subscripts are truncated to integers. Function identifiers are followed by parentheses, possibly enclosing arguments. The three types of identifiers do not conflict; a program can have a variable named **x**, an array named **x** and a function named **x**, all of which are separate and distinct.

2.3. Keywords

The following are reserved keywords:

ibase	**if**
obase	**break**
scale	**define**
sqrt	**auto**
length	**return**
while	**quit**
for	

2.4. Constants

Constants consist of arbitrarily long numbers with an optional decimal point. The hexadecimal digits **A**–**F** are also recognized as digits with values 10–15, respectively.

3. Expressions

The value of an expression is printed unless the main operator is an assignment. Precedence is the same as the order of presentation here, with highest appearing first. Left or right associativity, where applicable, is discussed with each operator.

3.1. Primitive expressions

3.1.1. Named expressions

Named expressions are places where values are stored. Simply stated, named expressions are legal on the left side of an assignment. The value of a named expression is the value stored in the place named.

3.1.1.1. *identifiers*

Simple identifiers are named expressions. They have an initial value of zero.

3.1.1.2. *array-name* [*expression*]

Array elements are named expressions. They have an initial value of zero.

3.1.1.3. scale, ibase and obase

The internal registers **scale**, **ibase** and **obase** are all named expressions. **scale** is the number of digits after the decimal point to be retained in arithmetic operations. **scale** has an initial value of zero. **ibase** and **obase** are the input and output number radix respectively. Both **ibase** and **obase** have initial values of 10.

3.1.2. Function calls

3.1.2.1. *function-name* ([*expression* [, *expression* ...]])

A function call consists of a function name followed by parentheses containing a comma-separated list of expressions, which are the function arguments. A whole array passed as an argument is specified by the array name followed by empty square brackets. All function arguments are passed by value. As a result, changes made to the formal parameters have no effect on the actual arguments. If the function terminates by executing a return statement, the value of the function is the value of the expression in the parentheses of the return statement or is zero if no expression is provided or if there is no return statement.

3.1.2.2. sqrt (*expression*)

The result is the square root of the expression. The result is truncated in the least significant decimal place. The scale of the result is the scale of the expression or the value of **scale**, whichever is larger.

3.1.2.3. length (*expression*)

The result is the total number of significant decimal digits in the expression. The scale of the result is zero.

3.1.2.4. scale (*expression*)

The result is the scale of the expression. The scale of the result is zero.

3.1.3. Constants

Constants are primitive expressions.

3.1.4. Parentheses

An expression surrounded by parentheses is a primitive expression. The parentheses are used to alter the normal precedence.

3.2. Unary operators

The unary operators bind right to left.

3.2.1. − *expression*

The result is the negative of the expression.

3.2.2. + + *named-expression*

The named expression is incremented by one. The result is the value of the named expression after incrementing.

3.2.3. − − *named-expression*

The named expression is decremented by one. The result is the value of the named expression after decrementing.

3.2.4. *named-expression* + +

The named expression is incremented by one. The result is the value of the named expression before incrementing.

3.2.5. *named-expression* − −

The named expression is decremented by one. The result is the value of the named expression before decrementing.

3.3. Exponentiation operator

The exponentiation operator binds right to left.

3.3.1. *expression* ^ *expression*

The result is the first expression raised to the power of the second expression. The second expression must be an integer. If a is the scale of the left expression and b is the absolute value of the right expression, then the scale of the result is:

$\min(a \times b, \max(\textbf{scale}, a))$

3.4. Multiplicative operators

The operators *, /, % bind left to right.

3.4.1. *expression* * *expression*

The result is the product of the two expressions. If a and b are the scales of the two expressions, then the scale of the result is:

$\min(a+b, \max(\textbf{scale}, a, b))$

3.4.2. *expression* / *expression*

The result is the quotient of the two expressions. The scale of the result is the value of **scale**.

3.4.3. *expression* % *expression*

The % operator produces the remainder of the division of the two expressions. More precisely, $a\%b$ is $a-a/b*b$.

The scale of the result is the sum of the scale of the divisor and the value of **scale**

3.5. Additive operators

The additive operators bind left to right.

3.5.1. *expression* + *expression*

The result is the sum of the two expressions. The scale of the result is the maximun of the scales of the expressions.

3.5.2. *expression* − *expression*

The result is the difference of the two expressions. The scale of the result is the maximum of the scales of the expressions.

3.6. assignment operators

The assignment operators bind right to left.

3.6.1. *named-expression* = *expression*

This expression results in assigning the value of the expression on the right to the named expression on the left.

3.6.2. *named-expression* =+ *expression*

3.6.3. *named-expression* =− *expression*

3.6.4. *named-expression* =* *expression*

3.6.5. *named-expression* =/ *expression*

3.6.6. *named-expression* =% *expression*

3.6.7. *named-expression* =^ *expression*

The result of the above expressions is equivalent to "named expression = named expression OP expression", where OP is the operator after the = sign.

4. Relations

Unlike all other operators, the relational operators are only valid as the object of an **if**, **while**, or inside a **for** statement.

4.1. *expression* < *expression*

4.2. *expression* > *expression*

4.3. *expression* <= *expression*

4.4. *expression* >= *expression*

4.5. *expression* == *expression*

4.6. *expression* != *expression*

5. Storage classes

There are only two storage classes in BC, global and automatic (local). Only identifiers that are to be local to a function need be declared with the **auto** command. The arguments to a function are local to the function. All other identifiers are assumed to be global and available to all functions. All identifiers, global and local, have initial values of zero. Identifiers declared as **auto** are allocated on entry to the function and released on returning from the function. They therefore do not retain values between function calls. **auto** arrays are specified by the array name followed by empty square brackets.

Automatic variables in BC do not work in exactly the same way as in either C or PL/I. On entry to a function, the old values of the names that appear as parameters and as automatic variables are pushed onto a stack. Until return is made from the function, reference to these names refers only to the new values.

6. Statements

Statements must be separated by semicolon or newline. Except where altered by control statements, execution is sequential.

6.1. Expression statements

When a statement is an expression, unless the main operator is an assignment, the value of the expression is printed, followed by a newline character.

6.2. Compound statements

Statements may be grouped together and used when one statement is expected by surrounding them with { }.

6.3. Quoted string statements

"any string"

This statement prints the string inside the quotes.

6.4. If statements

if (*relation*) *statement*

The substatement is executed if the relation is true.

6.5. While statements

while (*relation*) *statement*

The statement is executed while the relation is true. The test occurs before each execution of the statement.

6.6. For statements

for (*expression*; *relation*; *expression*) *statement*

The for statement is the same as

first-expression
while (*relation*) {
    statement
    last-expression
}

All three expressions must be present.

6.7. Break statements

break

break causes termination of a **for** or **while** statement.

6.8. Auto statements

auto *identifier* [,*identifier*]

The auto statement causes the values of the identifiers to be pushed down. The identifiers can be ordinary identifiers or array identifiers. Array identifiers are specified by following the array name by empty square brackets. The auto statement must be the first statement in a function definition.

6.9. Define statements

define ([*parameter* [,*parameter* . . .]]) {
statements }

The define statement defines a function. The parameters may be ordinary identifiers or array names. Array names must be followed by empty square brackets.

6.10. Return statements

return

return (*expression*)

The return statement causes termination of a function, popping of its auto variables, and specifies the result of the function. The first form is equivalent to **return** (**0**). The result of the function is the result of the expression in parentheses.

6.11. Quit

The quit statement stops execution of a BC program and returns control to UNIX when it is first encountered. Because it is not treated as an executable statement, it cannot be used in a function definition or in an **if, for,** or **while** statement.

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.

November 15, 1978

†UNIX is a Trademark of Bell Laboratories.

DC — An Interactive Desk Calculator

Robert Morris

Lorinda Cherry

Bell Laboratories
Murray Hill, New Jersey 07974

DC is an arbitrary precision arithmetic package implemented on the UNIX† time-sharing system in the form of an interactive desk calculator. It works like a stacking calculator using reverse Polish notation. Ordinarily DC operates on decimal integers, but one may specify an input base, output base, and a number of fractional digits to be maintained.

A language called BC [1] has been developed which accepts programs written in the familiar style of higher-level programming languages and compiles output which is interpreted by DC. Some of the commands described below were designed for the compiler interface and are not easy for a human user to manipulate.

Numbers that are typed into DC are put on a push-down stack. DC commands work by taking the top number or two off the stack, performing the desired operation, and pushing the result on the stack. If an argument is given, input is taken from that file until its end, then from the standard input.

SYNOPTIC DESCRIPTION

Here we describe the DC commands that are intended for use by people. The additional commands that are intended to be invoked by compiled output are described in the detailed description.

Any number of commands are permitted on a line. Blanks and new-line characters are ignored except within numbers and in places where a register name is expected.

The following constructions are recognized:

number

The value of the number is pushed onto the main stack. A number is an unbroken string of the digits 0-9 and the capital letters A−F which are treated as digits with values 10−15 respectively. The number may be preceded by an underscore to input a negative number. Numbers may contain decimal points.

+ − * % ^

The top two values on the stack are added (+), subtracted (−), multiplied (*), divided (/), remaindered (%), or exponentiated (^). The two entries are popped off the stack; the result is pushed on the stack in their place. The result of a division is an integer truncated toward zero. See the detailed description below for the treatment of numbers with decimal points. An exponent must not have any digits after the decimal point.

†UNIX is a Trademark of Bell Laboratories.

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

[2] K. C. Knowlton, *A Fast Storage Allocator,* Comm. ACM **8**, pp. 623-625 (Oct. 1965).

A Portable Fortran 77 Compiler

S. I. Feldman

P. J. Weinberger

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The Fortran language has just been revised. The new language, known as Fortran 77, became an official American National Standard on April 3, 1978. We report here on a compiler and run-time system for the new extended language. This is believed to be the first complete Fortran 77 system to be implemented. This compiler is designed to be portable, to be correct and complete, and to generate code compatible with calling sequences produced by C compilers. In particular, this Fortran is quite usable on UNIX† systems. In this paper, we describe the language compiled, interfaces between procedures, and file formats assumed by the I/O system. An appendix describes the Fortran 77 language.

1 August 1978

†UNIX is a Trademark of Bell Laboratories.

A Portable Fortran 77 Compiler

S. I. Feldman

P. J. Weinberger

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

The Fortran language has just been revised. The new language, known as Fortran 77, became an official American National Standard [1] on April 3, 1978. for the language, known as Fortran 77, is about to be published. Fortran 77 supplants 1966 Standard Fortran [2]. We report here on a compiler and run-time system for the new extended language. The compiler and computation library were written by SIF, the I/O system by PJW. We believe ours to be the first complete Fortran 77 system to be implemented. This compiler is designed to be portable to a number of different machines, to be correct and complete, and to generate code compatible with calling sequences produced by compilers for the C language [3]. In particular, it is in use on UNIX† systems. Two families of C compilers are in use at Bell Laboratories, those based on D. M. Ritchie's PDP-11 compiler[4] and those based on S. C. Johnson's portable C compiler [5]. This Fortran compiler can drive the second passes of either family. In this paper, we describe the language compiled, interfaces between procedures, and file formats assumed by the I/O system. We will describe implementation details in companion papers.

1.1. Usage

At present, versions of the compiler run on and compile for the PDP-11, the VAX-11/780, and the Interdata 8/32 UNIX systems. The command to run the compiler is

f77 *flags file . . .*

f77 is a general-purpose command for compiling and loading Fortran and Fortran-related files. EFL [6] and Ratfor [7] source files will be preprocessed before being presented to the Fortran compiler. C and assembler source files will be compiled by the appropriate programs. Object files will be loaded. (The **f77** and **cc** commands cause slightly different loading sequences to be generated, since Fortran programs need a few extra libraries and a different startup routine than do C programs.) The following file name suffixes are understood:

.f	Fortran source file
.e	EFL source file
.r	Ratfor source file
.c	C source file
.s	Assembler source file
.o	Object file

The following flags are understood:

–S	Generate assembler output for each source file, but do not assemble it. Assem-

†UNIX is a Trademark of Bell Laboratories.

bler output for a source file **x.f, x.e, x.r,** or **x.c** is put on file **x.s.**

−c	Compile but do not load. Output for **x.f, x.e, x.r, x.c,** or **x.s** is put on file **x.o.**
−m	Apply the M4 macro preprocessor to each EFL or Ratfor source file before using the appropriate compiler.
−f	Apply the EFL or Ratfor processor to all relevant files, and leave the output from **x.e** or **x.r** on **x.f.** Do not compile the resulting Fortran program.
−p	Generate code to produce usage profiles.
−o *f*	Put executable module on file *f.* (Default is **a.out**).
−w	Suppress all warning messages.
−w66	Suppress warnings about Fortran 66 features used.
−O	Invoke the C object code optimizer.
−C	Compile code the checks that subscripts are within array bounds.
−onetrip	Compile code that performs every **do** loop at least once. (see Section 2.10).
−U	Do not convert upper case letters to lower case. The default is to convert Fortran programs to lower case.
−u	Make the default type of a variable **undefined.** (see Section 2.3).
−I2	On machines which support short integers, make the default integer constants and variables short. (**−I4** is the standard value of this option). (see Section 2.14). All logical quantities will be short.
−E	The remaining characters in the argument are used as an EFL flag argument.
−R	The remaining characters in the argument are used as a Ratfor flag argument.
−F	Ratfor and and EFL source programs are pre-processed into Fortran files, but those files are not compiled or removed.

Other flags, all library names (arguments beginning −l), and any names not ending with one of the understood suffixes are passed to the loader.

1.2. Documentation Conventions

In running text, we write Fortran keywords and other literal strings in boldface lower case. Examples will be presented in lightface lower case. Names representing a class of values will be printed in italics.

1.3. Implementation Strategy

The compiler and library are written entirely in C. The compiler generates C compiler intermediate code. Since there are C compilers running on a variety of machines, relatively small changes will make this Fortran compiler generate code for any of them. Furthermore, this approach guarantees that the resulting programs are compatible with C usage. The runtime computational library is complete. The mathematical functions are computed to at least 63 bit precision. The runtime I/O library makes use of D. M. Ritchie's Standard C I/O package [8] for transferring data. With the few exceptions described below, only documented calls are used, so it should be relatively easy to modify to run on other operating systems.

2. LANGUAGE EXTENSIONS

Fortran 77 includes almost all of Fortran 66 as a subset. We describe the differences briefly in the Appendix. The most important additions are a character string data type, file-oriented input/output statements, and random access I/O. Also, the language has been cleaned up considerably.

In addition to implementing the language specified in the new Standard, our compiler implements a few extensions described in this section. Most are useful additions to the

language. The remainder are extensions to make it easier to communicate with C procedures or to permit compilation of old (1966 Standard) programs.

2.1. Double Complex Data Type

The new type **double complex** is defined. Each datum is represented by a pair of double precision real variables. A double complex version of every **complex** built-in function is provided. The specific function names begin with **z** instead of **c**.

2.2. Internal Files

The Fortran 77 standard introduces "internal files" (memory arrays), but restricts their use to formatted sequential I/O statements. Our I/O system also permits internal files to be used in direct and unformatted reads and writes.

2.3. Implicit Undefined statement

Fortran 66 has a fixed rule that the type of a variable that does not appear in a type statement is **integer** if its first letter is **i, j, k, l, m** or **n**, and **real** otherwise. Fortran 77 has an **implicit** statement for overriding this rule. As an aid to good programming practice, we permit an additional type, **undefined.** The statement

```
implicit undefined(a-z)
```

turns off the automatic data typing mechanism, and the compiler will issue a diagnostic for each variable that is used but does not appear in a type statement. Specifying the **−u** compiler flag is equivalent to beginning each procedure with this statement.

2.4. Recursion

Procedures may call themselves, directly or through a chain of other procedures.

2.5. Automatic Storage

Two new keywords are recognized, **static** and **automatic.** These keywords may appear as "types" in type statements and in **implicit** statements. Local variables are static by default; there is exactly one copy of the datum, and its value is retained between calls. There is one copy of each variable declared **automatic** for each invocation of the procedure. Automatic variables may not appear in **equivalence, data,** or **save** statements.

2.6. Source Input Format

The Standard expects input to the compiler to be in 72 column format: except in comment lines, the first five characters are the statement number, the next is the continuation character, and the next sixty-six are the body of the line. (If there are fewer than seventy-two characters on a line, the compiler pads it with blanks; characters after the seventy-second are ignored).

In order to make it easier to type Fortran programs, our compiler also accepts input in variable length lines. An ampersand ("&") in the first position of a line indicates a continuation line; the remaining characters form the body of the line. A tab character in one of the first six positions of a line signals the end of the statement number and continuation part of the line; the remaining characters form the body of the line. A tab elsewhere on the line is treated as another kind of blank by the compiler.

In the Standard, there are only 26 letters — Fortran is a one-case language. Consistent with ordinary UNIX system usage, our compiler expects lower case input. By default, the compiler converts all upper case characters to lower case except those inside character constants. However, if the **−U** compiler flag is specified, upper case letters are not transformed. In this mode, it is possible to specify external names with upper case letters in them, and to have distinct variables differing only in case. Regardless of the setting of

the flag, keywords will only be recognized in lower case.

2.7. Include Statement

The statement

```
include 'stuff'
```

is replaced by the contents of the file **stuff.** **includes** may be nested to a reasonable depth, currently ten.

2.8. Binary Initialization Constants

A **logical, real,** or **integer** variable may be initialized in a **data** statement by a binary constant, denoted by a letter followed by a quoted string. If the letter is **b**, the string is binary, and only zeroes and ones are permitted. If the letter is **o**, the string is octal, with digits **0–7**. If the letter is **z** or **x**, the string is hexadecimal, with digits **0–9**, **a–f**. Thus, the statements

```
integer a(3)
data a / b'1010', o'12', z'a' /
```

initialize all three elements of **a** to ten.

2.9. Character Strings

For compatibility with C usage, the following backslash escapes are recognized:

\n	newline
\t	tab
\b	backspace
\f	form feed
\0	null
\'	apostrophe (does not terminate a string)
\"	quotation mark (does not terminate a string)
\\	\
\x	x, where x is any other character

Fortran 77 only has one quoting character, the apostrophe. Our compiler and I/O system recognize both the apostrophe (') and the double-quote ("). If a string begins with one variety of quote mark, the other may be embedded within it without using the repeated quote or backslash escapes.

Every unequivalenced scalar local character variable and every character string constant is aligned on an **integer** word boundary. Each character string constant appearing outside a **data** statement is followed by a null character to ease communication with C routines.

2.10. Hollerith

Fortran 77 does not have the old Hollerith (*n***h**) notation, though the new Standard recommends implementing the old Hollerith feature in order to improve compatibility with old programs. In our compiler, Hollerith data may be used in place of character string constants, and may also be used to initialize non-character variables in **data** statements.

2.11. Equivalence Statements

As a very special and peculiar case, Fortran 66 permits an element of a multiply-dimensioned array to be represented by a singly-subscripted reference in **equivalence** statements. Fortran 77 does not permit this usage, since subscript lower bounds may now be different from 1. Our compiler permits single subscripts in **equivalence** statements, under the interpretation that all missing subscripts are equal to 1. A warning message is

printed for each such incomplete subscript.

2.12. One-Trip DO Loops

The Fortran 77 Standard requires that the range of a **do** loop not be performed if the initial value is already past the limit value, as in

```
do 10 i = 2, 1
```

The 1966 Standard stated that the effect of such a statement was undefined, but it was common practice that the range of a **do** loop would be performed at least once. In order to accommodate old programs, though they were in violation of the 1966 Standard, the **−onetrip** compiler flag causes non-standard loops to be generated.

2.13. Commas in Formatted Input

The I/O system attempts to be more lenient than the Standard when it seems worthwhile. When doing a formatted read of non-character variables, commas may be used as value separators in the input record, overriding the field lengths given in the format statement. Thus, the format

```
(i10, f20.10, i4)
```

will read the record

```
−345,.05e−3,12
```

correctly.

2.14. Short Integers

On machines that support halfword integers, the compiler accepts declarations of type **integer*2.** (Ordinary integers follow the Fortran rules about occupying the same space as a REAL variable; they are assumed to be of C type **long int**; halfword integers are of C type **short int.**) An expression involving only objects of type **integer*2** is of that type. Generic functions return short or long integers depending on the actual types of their arguments. If a procedure is compiled using the **−I2** flag, all small integer constants will be of type **integer*2.** If the precision of an integer-valued intrinsic function is not determined by the generic function rules, one will be chosen that returns the prevailing length (**integer*2** when the **−I2** command flag is in effect). When the **−I2** option is in effect, all quantities of type **logical** will be short. Note that these short integer and logical quantities do not obey the standard rules for storage association.

2.15. Additional Intrinsic Functions

This compiler supports all of the intrinsic functions specified in the Fortran 77 Standard. In addition, there are functions for performing bitwise Boolean operations (**or, and, xor,** and **not)** and for accessing the UNIX command arguments (**getarg** and **iargc**).

3. VIOLATIONS OF THE STANDARD

We know only thre ways in which our Fortran system violates the new standard:

3.1. Double Precision Alignment

The Fortran standards (both 1966 and 1977) permit **common** or **equivalence** statements to force a double precision quantity onto an odd word boundary, as in the following example:

```
real a(4)
double precision b,c

equivalence (a(1),b), (a(4),c)
```

Some machines (e.g., Honeywell 6000, IBM 360) require that double precision quantities be on double word boundaries; other machines (e.g., IBM 370), run inefficiently if this alignment rule is not observed. It is possible to tell which equivalenced and common variables suffer from a forced odd alignment, but every double precision argument would have to be assumed on a bad boundary. To load such a quantity on some machines, it would be necessary to use separate operations to move the upper and lower halves into the halves of an aligned temporary, then to load that double precision temporary; the reverse would be needed to store a result. We have chosen to require that all double precision real and complex quantities fall on even word boundaries on machines with corresponding hardware requirements, and to issue a diagnostic if the source code demands a violation of the rule.

3.2. Dummy Procedure Arguments

If any argument of a procedure is of type character, all dummy procedure arguments of that procedure must be declared in an **external** statement. This requirement arises as a subtle corollary of the way we represent character string arguments and of the one-pass nature of the compiler. A warning is printed if a dummy procedure is not declared **external.** Code is correct if there are no **character** arguments.

3.3. T and TL Formats

The implementation of the **t** (absolute tab) and **tl** (leftward tab) format codes is defective. These codes allow rereading or rewriting part of the record which has already been processed. (Section 6.3.2 in the Appendix.) The implementation uses seeks, so if the unit is not one which allows seeks, such as a terminal, the program is in error. (People who can make a case for using **tl** should let us know.) A benefit of the implementation chosen is that there is no upper limit on the length of a record, nor is it necessary to predeclare any record lengths except where specifically required by Fortran or the operating system.

4. INTER-PROCEDURE INTERFACE

To be able to write C procedures that call or are called by Fortran procedures, it is necessary to know the conventions for procedure names, data representation, return values, and argument lists that the compiled code obeys.

4.1. Procedure Names

On UNIX systems, the name of a common block or a Fortran procedure has an underscore appended to it by the compiler to distinguish it from a C procedure or external variable with the same user-assigned name. Fortran library procedure names have embedded underscores to avoid clashes with user-assigned subroutine names.

4.2. Data Representations

The following is a table of corresponding Fortran and C declarations:

Fortran	C
integer*2 x	short int x;
integer x	long int x;
logical x	long int x;
real x	float x;
double precision x	double x;
complex x	struct { float r, i; } x;
double complex x	struct { double dr, di; } x;
character*6 x	char x[6];

(By the rules of Fortran, **integer, logical,** and **real** data occupy the same amount of memory).

4.3. Return Values

A function of type **integer, logical, real,** or **double precision** declared as a C function that returns the corresponding type. A **complex** or **double complex** function is equivalent to a C routine with an additional initial argument that points to the place where the return value is to be stored. Thus,

```
complex function f( . . . )
```

is equivalent to

```
f_(temp, . . .)
struct { float r, i; } *temp;
. . .
```

A character-valued function is equivalent to a C routine with two extra initial arguments: a data address and a length. Thus,

```
character*15 function g( . . . )
```

is equivalent to

```
g_(result, length, . . .)
char result[ ];
long int length;
. . .
```

and could be invoked in C by

```
char chars[15];
. . .
g_(chars, 15L, . . . );
```

Subroutines are invoked as if they were **integer**-valued functions whose value specifies which alternate return to use. Alternate return arguments (statement labels) are not passed to the function, but are used to do an indexed branch in the calling procedure. (If the subroutine has no entry points with alternate return arguments, the returned value is undefined.) The statement

```
call nret(*1, *2, *3)
```

is treated exactly as if it were the computed **goto**

```
goto (1, 2, 3), nret( )
```

4.4. Argument Lists

All Fortran arguments are passed by address. In addition, for every argument that is of type character or that is a dummy procedure, an argument giving the length of the value is passed. (The string lengths are **long int** quantities passed by value). The order of arguments is then:

Extra arguments for complex and character functions
Address for each datum or function
A **long int** for each character or procedure argument

Thus, the call in

```
external f
character*7 s
integer b(3)
. . .
call sam(f, b(2), s)
```

is equivalent to that in

```
int f();
char s[7];
long int b[3];
. . .
sam_(f, &b[1], s, 0L, 7L);
```

Note that the first element of a C array always has subscript zero, but Fortran arrays begin at 1 by default. Fortran arrays are stored in column-major order, C arrays are stored in row-major order.

5. FILE FORMATS

5.1. Structure of Fortran Files

Fortran requires four kinds of external files: sequential formatted and unformatted, and direct formatted and unformatted. On UNIX systems, these are all implemented as ordinary files which are assumed to have the proper internal structure.

Fortran I/O is based on "records". When a direct file is opened in a Fortran program, the record length of the records must be given, and this is used by the Fortran I/O system to make the file look as if it is made up of records of the given length. In the special case that the record length is given as 1, the files are not considered to be divided into records, but are treated as byte-addressable byte strings; that is, as ordinary UNIX file system files. (A read or write request on such a file keeps consuming bytes until satisfied, rather than being restricted to a single record.)

The peculiar requirements on sequential unformatted files make it unlikely that they will ever be read or written by any means except Fortran I/O statements. Each record is preceded and followed by an integer containing the record's length in bytes.

The Fortran I/O system breaks sequential formatted files into records while reading by using each newline as a record separator. The result of reading off the end of a record is undefined according to the Standard. The I/O system is permissive and treats the record as being extended by blanks. On output, the I/O system will write a newline at the end of each record. It is also possible for programs to write newlines for themselves. This is an error, but the only effect will be that the single record the user thought he wrote will be treated as more than one record when being read or backspaced over.

5.2. Portability Considerations

The Fortran I/O system uses only the facilities of the standard C I/O library, a widely available and fairly portable package, with the following two nonstandard features: The I/O system needs to know whether a file can be used for direct I/O, and whether or not it is possible to backspace. Both of these facilities are implemented using the **fseek** routine, so there is a routine **canseek** which determines if **fseek** will have the desired effect. Also, the **inquire** statement provides the user with the ability to find out if two files are the same, and to get the name of an already opened file in a form which would enable the program to reopen it. (The UNIX operating system implementation attempts to determine the full pathname.) Therefore there are two routines which depend on facilities of the operating system to provide these two services. In any case, the I/O system runs on the PDP-11, VAX-11/780, and Interdata 8/32 UNIX systems.

5.3. Pre-Connected Files and File Positions

Units 5, 6, and 0 are preconnected when the program starts. Unit 5 is connected to the standard input, unit 6 is connected to the standard output, and unit 0 is connected to the standard error unit. All are connected for sequential formatted I/O.

All the other units are also preconnected when execution begins. Unit *n* is connected to a file named **fort.***n*. These files need not exist, nor will they be created unless their units are used without first executing an **open**. The default connection is for sequential formatted I/O.

The Standard does not specify where a file which has been explicitly **open**ed for sequential I/O is initially positioned. In fact, the I/O system attempts to position the file at the end, so a **write** will append to the file and a **read** will result in an end-of-file indication. To position a file to its beginning, use a **rewind** statement. The preconnected units 0, 5, and 6 are positioned as they come from the program's parent process.

REFERENCES

1. *Sigplan Notices* **11**, No.3 (1976), as amended in X3J3 internal documents through "/90.1".
2. *USA Standard FORTRAN, USAS X3.9-1966*, New York: United States of America Standards Institute, March 7, 1966. Clarified in *Comm. ACM* **12**, 289 (1969) and *Comm. ACM* **14**, 628 (1971).
3. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs: Prentice-Hall (1978).
4. D. M. Ritchie, private communication.
5. S. C. Johnson, "A Portable Compiler: Theory and Practice", Proc. 5th ACM Symp. on Principles of Programming Languages (January 1978).
6. S. I. Feldman, "An Informal Description of EFL", internal memorandum.
7. B. W. Kernighan, "RATFOR — A Preprocessor for a Rational Fortran", *Bell Laboratories Computing Science Technical Report #55*, (January 1977).
8. D. M. Ritchie, private communication.

APPENDIX. Differences Between Fortran 66 and Fortran 77

The following is a very brief description of the differences between the 1966 [2] and the 1977 [1] Standard languages. We assume that the reader is familiar with Fortran 66. We do not pretend to be complete, precise, or unbiased, but plan to describe what we feel are the most important aspects of the new language. At present the only current information on the 1977 Standard is in publications of the X3J3 Subcommittee of the American National Standards Institute. The following information is from the "/92" document. This draft Standard is written in English rather than a meta-language, but it is forbidding and legalistic. No tutorials or textbooks are available yet.

1. Features Deleted from Fortran 66

1.1. Hollerith

All notions of "Hollerith" (*n***h**) as data have been officially removed, although our compiler, like almost all in the foreseeable future, will continue to support this archaism.

1.2. Extended Range

In Fortran 66, under a set of very restrictive and rarely-understood conditions, it is permissible to jump out of the range of a **do** loop, then jump back into it. Extended range has been removed in the Fortran 77 language. The restrictions are so special, and the implementation of extended range is so unreliable in many compilers, that this change really counts as no loss.

2. Program Form

2.1. Blank Lines

Completely blank lines are now legal comment lines.

2.2. Program and Block Data Statements

A main program may now begin with a statement that gives that program an external name:

```
program work
```

Block data procedures may also have names.

```
block data stuff
```

There is now a rule that only *one* unnamed block data procedure may appear in a program. (This rule is not enforced by our system.) The Standard does not specify the effect of the program and block data names, but they are clearly intended to aid conventional loaders.

2.3. ENTRY Statement

Multiple entry points are now legal. Subroutine and function subprograms may have additional entry points, declared by an **entry** statement with an optional argument list.

```
entry extra(a, b, c)
```

Execution begins at the first statement following the **entry** line. All variable declarations must precede all executable statements in the procedure. If the procedure begins with a **subroutine** statement, all entry points are subroutine names. If it begins with a **function** statement, each entry is a function entry point, with type determined by the type declared for the entry name. If any entry is a character-valued function, then all entries must be. In a function, an entry name of the same type as that where control entered must be assigned a value. Arguments do not retain their values between calls. (The ancient trick

of calling one entry point with a large number of arguments to cause the procedure to "remember" the locations of those arguments, then invoking an entry with just a few arguments for later calculation, is still illegal. Furthermore, the trick doesn't work in our implementation, since arguments are not kept in static storage.)

2.4. DO Loops

do variables and range parameters may now be of integer, real, or double precision types. (The use of floating point **do** variables is very dangerous because of the possibility of unexpected roundoff, and we strongly recommend against their use). The action of the **do** statement is now defined for all values of the **do** parameters. The statement

```
do 10 i = l, u, d
```

performs $\max(0, \lfloor (u-l)/d \rfloor)$ iterations. The **do** variable has a predictable value when exiting a loop: the value at the time a **goto** or **return** terminates the loop; otherwise the value that failed the limit test.

2.5. Alternate Returns

In a **subroutine** or subroutine **entry** statement, some of the arguments may be noted by an asterisk, as in

```
subroutine s(a, *, b, *)
```

The meaning of the "alternate returns" is described in section 5.2 of the Appendix.

3. Declarations

3.1. CHARACTER Data Type

One of the biggest improvements to the language is the addition of a character-string data type. Local and common character variables must have a length denoted by a constant expression:

```
character*17 a, b(3,4)
character*(6+3) c
```

If the length is omitted entirely, it is assumed equal to 1. A character string argument may have a constant length, or the length may be declared to be the same as that of the corresponding actual argument at run time by a statement like

```
character*(*) a
```

(There is an intrinsic function **len** that returns the actual length of a character string). Character arrays and common blocks containing character variables must be packed: in an array of character variables, the first character of one element must follow the last character of the preceding element, without holes.

3.2. IMPLICIT Statement

The traditional implied declaration rules still hold: a variable whose name begins with **i, j, k, l, m,** or **n** is of type **integer,** other variables are of type **real,** unless otherwise declared. This general rule may be overridden with an **implicit** statement:

```
implicit real(a-c,g), complex(w-z), character*(17) (s)
```

declares that variables whose name begins with an **a ,b, c,** or **g** are **real,** those beginning with **w, x, y,** or **z** are assumed **complex,** and so on. It is still poor practice to depend on implicit typing, but this statement is an industry standard.

3.3. PARAMETER Statement

It is now possible to give a constant a symbolic name, as in

```
parameter (x=17, y=x/3, pi=3.14159d0, s='hello')
```

The type of each parameter name is governed by the same implicit and explicit rules as for a variable. The right side of each equal sign must be a constant expression (an expression made up of constants, operators, and already defined parameters).

3.4. Array Declarations

Arrays may now have as many as seven dimensions. (Only three were permitted in 1966). The lower bound of each dimension may be declared to be other than 1 by using a colon. Furthermore, an adjustable array bound may be an integer expression involving constants, arguments, and variables in **common.**

```
real a(-5:3, 7, m:n), b(n+1:2*n)
```

The upper bound on the last dimension of an array argument may be denoted by an asterisk to indicate that the upper bound is not specified:

```
integer a(5, *), b(*), c(0:1, -2:*)
```

3.5. SAVE Statement

A poorly known rule of Fortran 66 is that local variables in a procedure do not necessarily retain their values between invocations of that procedure. At any instant in the execution of a program, if a common block is declared neither in the currently executing procedure nor in any of the procedures in the chain of callers, all of the variables in that common block also become undefined. (The only exceptions are variables that have been defined in a **data** statement and never changed). These rules permit overlay and stack implementations for the affected variables. Fortran 77 permits one to specify that certain variables and common blocks are to retain their values between invocations. The declaration

```
save a, /b/, c
```

leaves the values of the variables **a** and **c** and all of the contents of common block **b** unaffected by a return. The simple declaration

```
save
```

has this effect on all variables and common blocks in the procedure. A common block must be **saved** in every procedure in which it is declared if the desired effect is to occur.

3.6. INTRINSIC Statement

All of the functions specified in the Standard are in a single category, "intrinsic functions", rather than being divided into "intrinsic" and "basic external" functions. If an intrinsic function is to be passed to another procedure, it must be declared **intrinsic.** Declaring it **external** (as in Fortran 66) causes a function other than the built-in one to be passed.

4. Expressions

4.1. Character Constants

Character string constants are marked by strings surrounded by apostrophes. If an apostrophe is to be included in a constant, it is repeated:

```
'abc'
'ain''t'
```

There are no null (zero-length) character strings in Fortran 77. Our compiler has two different quotation marks, " ' " and " " ". (See Section 2.9 in the main text.)

4.2. Concatenation

One new operator has been added, character string concatenation, marked by a double slash ("//"). The result of a concatenation is the string containing the characters of the left operand followed by the characters of the right operand. The strings

```
'ab' // 'cd'
'abcd'
```

are equal. The strings being concatenated must be of constant length in all concatenations that are not the right sides of assignments. (The only concatenation expressions in which a character string declared adjustable with a "*(*)" modifier or a substring denotation with nonconstant position values may appear are the right sides of assignments).

4.3. Character String Assignment

The left and right sides of a character assignment may not share storage. (The assumed implementation of character assignment is to copy characters from the right to the left side.) If the left side is longer than the right, it is padded with blanks. If the left side is shorter than the right, trailing characters are discarded.

4.4. Substrings

It is possible to extract a substring of a character variable or character array element, using the colon notation:

```
a(i,j) (m:n)
```

is the string of $(n-m+1)$ characters beginning at the m^{th} character of the character array element a_{ij}. Results are undefined unless $m \leqslant n$. Substrings may be used on the left sides of assignments and as procedure actual arguments.

4.5. Exponentiation

It is now permissible to raise real quantities to complex powers, or complex quantities to real or complex powers. (The principal part of the logarithm is used). Also, multiple exponentiation is now defined:

```
a**b**c = a ** (b**c)
```

4.6. Relaxation of Restrictions

Mixed mode expressions are now permitted. (For instance, it is permissible to combine integer and complex quantities in an expression.)

Constant expressions are permitted where a constant is allowed, except in **data** statements. (A constant expression is made up of explicit constants and **parameters** and the Fortran operators, except for exponentiation to a floating-point power). An adjustable dimension may now be an integer expression involving constants, arguments, and variables in B common..

Subscripts may now be general integer expressions; the old $cv \pm c'$ rules have been removed. **do** loop bounds may be general integer, real, or double precision expressions. Computed **goto** expressions and I/O unit numbers may be general integer expressions.

5. Executable Statements

5.1. IF-THEN-ELSE

At last, the if-then-else branching structure has been added to Fortran. It is called a "Block If". A Block If begins with a statement of the form

```
if ( . . . ) then
```

and ends with an

```
end if
```

statement. Two other new statements may appear in a Block If. There may be several

```
else if(. . .) then
```

statements, followed by at most one

```
else
```

statement. If the logical expression in the Block If statement is true, the statements following it up to the next **elseif, else,** or **endif** are executed. Otherwise, the next **elseif** statement in the group is executed. If none of the **elseif** conditions are true, control passes to the statements following the **else** statement, if any. (The **else** must follow all **elseifs** in a Block If. Of course, there may be Block Ifs embedded inside of other Block If structures). A case construct may be rendered

```
if (s .eq. 'ab') then
 . . .
else if (s .eq. 'cd') then
 . . .
else
 . . .
end if
```

5.2. Alternate Returns

Some of the arguments of a subroutine call may be statement labels preceded by an asterisk, as in

```
call joe(j, *10, m, *2)
```

A **return** statement may have an integer expression, such as

```
return k
```

If the entry point has n alternate return (asterisk) arguments and if $1 \leqslant k \leqslant n$, the return is followed by a branch to the corresponding statement label; otherwise the usual return to the statement following the **call** is executed.

6. Input/Output

6.1. Format Variables

A format may be the value of a character expression (constant or otherwise), or be stored in a character array, as in

```
write(6, '(i5)') x
```

6.2. END=, ERR=, and IOSTAT= Clauses

A **read** or **write** statement may contain **end=**, **err=**, and **iostat=** clauses, as in

```
write(6, 101, err=20, iostat=a(4))
read(5, 101, err=20, end=30, iostat=x)
```

Here 5 and 6 are the *units* on which the I/O is done, 101 is the statement number of the associated format, 20 and 30 are statement numbers, and **a** and **x** are integers. If an error occurs during I/O, control returns to the program at statement 20. If the end of the file is reached, control returns to the program at statement 30. In any case, the variable referred to in the **iostat=** clause is given a value when the I/O statement finishes. (Yes, the value is assigned to the name on the right side of the equal sign.) This value is zero if all went well, negative for end of file, and some positive value for errors.

6.3. Formatted I/O

6.3.1. Character Constants

Character constants in formats are copied literally to the output. Character constants cannot be read into.

```
write(6,'(i2," isn""t ",i1)') 7, 4
```

produces

```
 7 isn't 4
```

Here the format is the character constant

```
(i2,' isn''t ',i1)
```

and the character constant

```
 isn't
```

is copied into the output.

6.3.2. Positional Editing Codes

t, tl, tr, and **x** codes control where the next character is in the record. **tr***n* or *n***x** specifies that the next character is *n* to the right of the current position. **tl***n* specifies that the next character is *n* to the left of the current position, allowing parts of the record to be reconsidered. **t***n* says that the next character is to be character number *n* in the record. (See section 3.4 in the main text.)

6.3.3. Colon

A colon in the format terminates the I/O operation if there are no more data items in the I/O list, otherwise it has no effect. In the fragment

```
x='("hello", :, " there", i4)'
write(6, x) 12
write(6, x)
```

the first **write** statement prints **hello there 12**, while the second only prints **hello**.

6.3.4. Optional Plus Signs

According to the Standard, each implementation has the option of putting plus signs in front of non-negative numeric output. The **sp** format code may be used to make the optional plus signs actually appear for all subsequent items while the format is active. The **ss** format code guarantees that the I/O system will not insert the optional plus signs, and the **s** format code restores the default behavior of the I/O system. (Since we never put

out optional plus signs, ss and s codes have the same effect in our implementation.)

6.3.5. Blanks on Input

Blanks in numeric input fields, other than leading blanks will be ignored following a **bn** code in a format statement, and will be treated as zeros following a **bz** code in a format statement. The default for a unit may be changed by using the **open** statement. (Blanks are ignored by default.)

6.3.6. Unrepresentable Values

The Standard requires that if a numeric item cannot be represented in the form required by a format code, the output field must be filled with asterisks. (We think this should have been an option.)

6.3.7. Iw.m

There is a new integer output code, **i***w.m*. It is the same as **i***w*, except that there will be at least *m* digits in the output field, including, if necessary, leading zeros. The case **i***w*.0 is special, in that if the value being printed is 0, the output field is entirely blank. **i***w*.**1** is the same as **i***w*.

6.3.8. Floating Point

On input, exponents may start with the letter **E, D, e,** or **d.** All have the same meaning. On output we always use **e**. The **e** and **d** format codes also have identical meanings. A leading zero before the decimal point in **e** output without a scale factor is optional with the implementation. (We do not print it.) There is a **g***w.d* format code which is the same as **e***w.d* and **f***w.d* on input, but which chooses **f** or **e** formats for output depending. on the size of the number and of *d*.

6.3.9. "A" Format Code

A codes are used for character values. **a***w* use a field width of *w*, while a plain **a** uses the length of the character item.

6.4. Standard Units

There are default formatted input and output units. The statement

```
read 10, a, b
```

reads from the standard unit using format statement 10. The default unit may be explicitly specified by an asterisk, as in

```
read(*, 10) a,b
```

Similarly, the standard output units is specified by a **print** statement or an asterisk unit:

```
print 10
write(*, 10)
```

6.5. List-Directed Formatting

List-directed I/O is a kind of free form input for sequential I/O. It is invoked by using an asterisk as the format identifier, as in

```
read(6, *) a,b,c
```

On input, values are separated by strings of blanks and possibly a comma. Values, except for character strings, cannot contain blanks. End of record counts as a blank, except in character strings, where it is ignored. Complex constants are given as two real constants separated by a comma and enclosed in parentheses. A null input field, such as between two consecutive commas, means the corresponding variable in the I/O list is not changed. Values may be preceded by repetition counts, as in

```
4*(3.,2.)  2*, 4*'hello'
```

which stands for 4 complex constants, 2 null values, and 4 string constants.

For output, suitable formats are chosen for each item. The values of character strings are printed; they are not enclosed in quotes, so they cannot be read back using list-directed input.

6.6. Direct I/O

A file connected for direct access consists of a set of equal-sized records each of which is uniquely identified by a positive integer. The records may be written or read in any order, using direct access I/O statements.

Direct access **read** and **write** statements have an extra argument, **rec=**, which gives the record number to be read or written.

```
read(2, rec=13, err=20) (a(i), i=1, 203)
```

reads the thirteenth record into the array **a.**

The size of the records must be given by an **open** statement (see below). Direct access files may be connected for either formatted or unformatted I/O.

6.7. Internal Files

Internal files are character string objects, such as variables or substrings, or arrays of type character. In the former cases there is only a single record in the file, in the latter case each array element is a record. The Standard includes only sequential formatted I/O on internal files. (I/O is not a very precise term to use here, but internal files are dealt with using **read** and **write**). There is no list-directed I/O on internal files. Internal files are used by giving the name of the character object in place of the unit number, as in

```
character*80 x
read(5,"(a)") x
read(x,"(i3,i4)") n1,n2
```

which reads a card image into **x** and then reads two integers from the front of it. A sequential **read** or **write** always starts at the beginning of an internal file.

(We also support a compatible extension, direct I/O on internal files. This is like direct I/O on external files, except that the number of records in the file cannot be changed.)

6.8. OPEN, CLOSE, and INQUIRE Statements

These statements are used to connect and disconnect units and files, and to gather information about units and files.

6.8.1. OPEN

The **open** statement is used to connect a file with a unit, or to alter some properties of the connection. The following is a minimal example.

```
open(1, file='fort.junk')
```

open takes a variety of arguments with meanings described below.

unit= a small non-negative integer which is the unit to which the file is to be connected. We allow, at the time of this writing, 0 through 9. If this parameter is the first one in the **open** statement, the **unit=** can be omitted.

iostat= is the same as in **read** or **write.**

err= is the same as in **read** or **write.**

file= a character expression, which when stripped of trailing blanks, is the name of the file to be connected to the unit. The filename should not be given if the **status=scratch.**

status= one of **old, new, scratch,** or **unknown.** If this parameter is not given, **unknown** is assumed. If **scratch** is given, a temporary file will be created. Temporary files are destroyed at the end of execution. If **new** is given, the file will be created if it doesn't exist, or truncated if it does. The meaning of **unknown** is processor dependent; our system treats it as synonymous with **old.**

access= sequential or **direct,** depending on whether the file is to be opened for sequential or direct I/O.

form= formatted or **unformatted.**

recl= a positive integer specifying the record length of the direct access file being opened. We measure all record lengths in bytes. On UNIX systems a record length of 1 has the special meaning explained in section 5.1 of the text.

blank= null or **zero.** This parameter has meaning only for formatted I/O. The default value is **null. zero** means that blanks, other than leading blanks, in numeric input fields are to be treated as zeros.

Opening a new file on a unit which is already connected has the effect of first closing the old file.

6.8.2. CLOSE

close severs the connection between a unit and a file. The unit number must be given. The optional parameters are **iostat=** and **err=** with their usual meanings, and **status=** either **keep** or **delete.** Scratch files cannot be kept, otherwise **keep** is the default. **delete** means the file will be removed. A simple example is

```
close(3, err=17)
```

6.8.3. INQUIRE

The **inquire** statement gives information about a unit ("inquire by unit") or a file ("inquire by file"). Simple examples are:

```
inquire(unit=3, namexx)
inquire(file='junk', number=n, exist=l)
```

file= a character variable specifies the file the **inquire** is about. Trailing blanks in the file name are ignored.

unit= an integer variable specifies the unit the **inquire** is about. Exactly one of **file=** or **unit=** must be used.

iostat=, err= are as before.

exist= a logical variable. The logical variable is set to **.true.** if the file or unit exists and is set to **.false.** otherwise.

opened= a logical variable. The logical variable is set to **.true.** if the file is connected to a unit or if the unit is connected to a file, and it is set to **.false.** otherwise.

number= an integer variable to which is assigned the number of the unit connected to the file, if any.

named= a logical variable to which is assigned **.true.** if the file has a name, or **.false.** otherwise.

name= a character variable to which is assigned the name of the file (inquire by file) or the name of the file connected to the unit (inquire by unit). The name will be the full name of the file.

access= a character variable to which will be assigned the value **'sequential'** if the connection is for sequential I/O, **'direct'** if the connection is for direct I/O. The value becomes undefined if there is no connection.

sequential= a character variable to which is assigned the value **'yes'** if the file could be connected for sequential I/O, **'no'** if the file could not be connected for sequential I/O, and **'unknown'** if we can't tell.

direct= a character variable to which is assigned the value **'yes'** if the file could be connected for direct I/O, **'no'** if the file could not be connected for direct I/O, and **'unknown'** if we can't tell.

form= a character variable to which is assigned the value **'formatted'** if the file is connected for formatted I/O, or **'unformatted'** if the file is connected for unformatted I/O.

formatted= a character variable to which is assigned the value **'yes'** if the file could be connected for formatted I/O, **'no'** if the file could not be connected for formatted I/O, and **'unknown'** if we can't tell.

unformatted= a character variable to which is assigned the value **'yes'** if the file could be connected for unformatted I/O, **'no'** if the file could not be connected for unformatted I/O, and **'unknown'** if we can't tell.

recl= an integer variable to which is assigned the record length of the records in the file if the file is connected for direct access.

nextrec= an integer variable to which is assigned one more than the number of the the last record read from a file connected for direct access.

blank= a character variable to which is assigned the value **'null'** if null blank control is in effect for the file connected for formatted I/O, **'zero'** if blanks are being converted to zeros and the file is connected for formatted I/O.

The gentle reader will remember that the people who wrote the standard probably weren't thinking of his needs. Here is an example. The declarations are omitted.

```
open(1, file="/dev/console")
```

On a UNIX system this statement opens the console for formatted sequential I/O. An **inquire** statement for either unit 1 or file "/dev/console" would reveal that the file exists, is connected to unit 1, has a name, namely "/dev/console", is opened for sequential I/O, could be connected for sequential I/O, could not be connected for direct I/O (can't seek), is connected for formatted I/O, could be connected for formatted I/O, could not be connected for unformatted I/O (can't seek), has neither a record length nor a next record number, and is ignoring blanks in numeric fields.

In the UNIX system environment, the only way to discover what permissions you have for a file is to open it and try to read and write it. The **err=** parameter will return system error numbers. The **inquire** statement does not give a way of determining permissions.

RATFOR — A Preprocessor for a Rational Fortran

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Although Fortran is not a pleasant language to use, it does have the advantages of universality and (usually) relative efficiency. The Ratfor language attempts to conceal the main deficiencies of Fortran while retaining its desirable qualities, by providing decent control flow statements:

- statement grouping
- **if-else** and **switch** for decision-making
- **while**, **for**, **do**, and **repeat-until** for looping
- **break** and **next** for controlling loop exits

and some "syntactic sugar":

- free form input (multiple statements/line, automatic continuation)
- unobtrusive comment convention
- translation of >, >=, etc., into .GT., .GE., etc.
- **return**(expression) statement for functions
- **define** statement for symbolic parameters
- **include** statement for including source files

Ratfor is implemented as a preprocessor which translates this language into Fortran.

Once the control flow and cosmetic deficiencies of Fortran are hidden, the resulting language is remarkably pleasant to use. Ratfor programs are markedly easier to write, and to read, and thus easier to debug, maintain and modify than their Fortran equivalents.

It is readily possible to write Ratfor programs which are portable to other environments. Ratfor is written in itself in this way, so it is also portable; versions of Ratfor are now running on at least two dozen different types of computers at over five hundred locations.

This paper discusses design criteria for a Fortran preprocessor, the Ratfor language and its implementation, and user experience.

RATFOR — A Preprocessor for a Rational Fortran

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

Most programmers will agree that Fortran is an unpleasant language to program in, yet there are many occasions when they are forced to use it. For example, Fortran is often the only language thoroughly supported on the local computer. Indeed, it is the closest thing to a universal programming language currently available: with care it is possible to write large, truly portable Fortran programs[1]. Finally, Fortran is often the most "efficient" language available, particularly for programs requiring much computation.

But Fortran *is* unpleasant. Perhaps the worst deficiency is in the control flow statements — conditional branches and loops — which express the logic of the program. The conditional statements in Fortran are primitive. The Arithmetic IF forces the user into at least two statement numbers and two (implied) GOTO's; it leads to unintelligible code, and is eschewed by good programmers. The Logical IF is better, in that the test part can be stated clearly, but hopelessly restrictive because the statement that follows the IF can only be one Fortran statement (with some *further* restrictions!). And of course there can be no ELSE part to a Fortran IF: there is no way to specify an alternative action if the IF is not satisfied.

The Fortran DO restricts the user to going forward in an arithmetic progression. It is fine for "1 to N in steps of 1 (or 2 or ...)", but there is no direct way to go backwards, or even (in ANSI Fortran[2]) to go from 1 to N−1. And of course the DO is useless if one's problem doesn't map into an arithmetic progression.

The result of these failings is that Fortran programs must be written with numerous labels and branches. The resulting code is particularly difficult to read and understand, and thus hard to debug and modify.

When one is faced with an unpleasant language, a useful technique is to define a new language that overcomes the deficiencies, and to translate it into the unpleasant one with a preprocessor. This is the approach taken with Ratfor. (The preprocessor idea is of course not new, and preprocessors for Fortran are especially popular today. A recent listing [3] of preprocessors shows more than 50, of which at least half a dozen are widely available.)

2. LANGUAGE DESCRIPTION

Design

Ratfor attempts to retain the merits of Fortran (universality, portability, efficiency) while hiding the worst Fortran inadequacies. The language *is* Fortran except for two aspects. First, since control flow is central to any program, regardless of the specific application, the primary task of Ratfor is to conceal this part of Fortran from the user, by providing decent control flow structures. These structures are sufficient and comfortable for structured programming in the narrow sense of programming without GOTO's. Second, since the preprocessor must examine an entire program to translate the control structure, it is possible at the same time to clean up many of the "cosmetic" deficiencies of Fortran, and thus provide a language which is easier and more pleasant to read and write.

Beyond these two aspects — control flow and cosmetics — Ratfor does nothing about the host of other weaknesses of Fortran. Although it would be straightforward to extend it to provide character strings, for example, they are not needed by everyone, and of course the preprocessor would be harder to implement. Throughout, the design principle which has determined what should be in Ratfor and what should not has been *Ratfor doesn't know any Fortran*. Any language feature which would require

This paper is a revised and expanded version of oe published in *Software—Practice and Experience*, October 1975. The Ratfor described here is the one in use on UNIX and GCOS at Bell Laboratories, Murray Hill, N. J.

that Ratfor really understand Fortran has been omitted. We will return to this point in the section on implementation.

Even within the confines of control flow and cosmetics, we have attempted to be selective in what features to provide. The intent has been to provide a small set of the most useful constructs, rather than to throw in everything that has ever been thought useful by someone.

The rest of this section contains an informal description of the Ratfor language. The control flow aspects will be quite familiar to readers used to languages like Algol, PL/I, Pascal, etc., and the cosmetic changes are equally straightforward. We shall concentrate on showing what the language looks like.

Statement Grouping

Fortran provides no way to group statements together, short of making them into a subroutine. The standard construction "if a condition is true, do this group of things," for example,

```
if (x > 100)
        { call error("x>100"); err = 1; return }
```

cannot be written directly in Fortran. Instead a programmer is forced to translate this relatively clear thought into murky Fortran, by stating the negative condition and branching around the group of statements:

```
        if (x .le. 100) goto 10
                call error(5hx>100)
                err = 1
                return
10      ...
```

When the program doesn't work, or when it must be modified, this must be translated back into a clearer form before one can be sure what it does.

Ratfor eliminates this error-prone and confusing back-and-forth translation; the first form *is* the way the computation is written in Ratfor. A group of statements can be treated as a unit by enclosing them in the braces { and }. This is true throughout the language: wherever a single Ratfor statement can be used, there can be several enclosed in braces. (Braces seem clearer and less obtrusive than **begin** and **end** or **do** and **end**, and of course **do** and **end** already have Fortran meanings.)

Cosmetics contribute to the readability of code, and thus to its understandability. The character ">" is clearer than ".GT.", so Ratfor translates it appropriately, along with several other similar shorthands. Although many Fortran compilers permit character strings in quotes (like "**x>100**"), quotes are not allowed in ANSI Fortran, so Ratfor converts it into the right number of **H**'s: computers count better than people do.

Ratfor is a free-form language: statements may appear anywhere on a line, and several may appear on one line if they are separated by semicolons. The example above could also be written as

```
if (x > 100) {
        call error("x>100")
        err = 1
        return
}
```

In this case, no semicolon is needed at the end of each line because Ratfor assumes there is one statement per line unless told otherwise.

Of course, if the statement that follows the **if** is a single statement (Ratfor or otherwise), no braces are needed:

```
if (y <= 0.0 & z <= 0.0)
        write(6, 20) y, z
```

No continuation need be indicated because the statement is clearly not finished on the first line. In general Ratfor continues lines when it seems obvious that they are not yet done. (The continuation convention is discussed in detail later.)

Although a free-form language permits wide latitude in formatting styles, it is wise to pick one that is readable, then stick to it. In particular, proper indentation is vital, to make the logical structure of the program obvious to the reader.

The "else" Clause

Ratfor provides an **else** statement to handle the construction "if a condition is true, do this thing, *otherwise* do that thing."

```
if (a <= b)
        { sw = 0; write(6, 1) a, b }
else
        { sw = 1; write(6, 1) b, a }
```

This writes out the smaller of **a** and **b**, then the larger, and sets **sw** appropriately.

The Fortran equivalent of this code is circuitous indeed:

```
      if (a .gt. b) goto 10
            sw = 0
            write(6, 1) a, b
            goto 20
10    sw = 1
      write(6, 1) b, a
20    ...
```

This is a mechanical translation; shorter forms exist, as they do for many similar situations. But all translations suffer from the same problem: since they are translations, they are less clear and understandable than code that is not a translation. To understand the Fortran version, one must scan the entire program to make sure that no other statement branches to statements 10 or 20 before one knows that indeed this is an **if-else** construction. With the Ratfor version, there is no question about how one gets to the parts of the statement. The **if-else** is a single unit, which can be read, understood, and ignored if not relevant. The program says what it means.

As before, if the statement following an **if** or an **else** is a single statement, no braces are needed:

```
if (a <= b)
      sw = 0
else
      sw = 1
```

The syntax of the **if** statement is

```
if (legal Fortran condition)
      Ratfor statement
else
      Ratfor statement
```

where the **else** part is optional. The *legal Fortran condition* is anything that can legally go into a Fortran Logical IF. Ratfor does not check this clause, since it does not know enough Fortran to know what is permitted. The *Ratfor statement* is any Ratfor or Fortran statement, or any collection of them in braces.

Nested if's

Since the statement that follows an **if** or an **else** can be any Ratfor statement, this leads immediately to the possibility of another **if** or **else**. As a useful example, consider this problem: the variable **f** is to be set to -1 if **x** is less than zero, to $+1$ if **x** is greater than 100, and to 0 otherwise. Then in Ratfor, we write

```
if (x < 0)
      f = -1
else if (x > 100)
      f = +1
else
      f = 0
```

Here the statement after the first **else** is another **if-else**. Logically it is just a single statement, although it is rather complicated.

This code says what it means. Any version written in straight Fortran will necessarily be indirect because Fortran does not let you say what you mean. And as always, clever shortcuts may turn out to be too clever to understand a year from now.

Following an **else** with an **if** is one way to write a multi-way branch in Ratfor. In general the structure

```
if (...)
      - - -
else if (...)
      - - -
else if (...)
      - - -
...
else
      - - -
```

provides a way to specify the choice of exactly one of several alternatives. (Ratfor also provides a **switch** statement which does the same job in certain special cases; in more general situations, we have to make do with spare parts.) The tests are laid out in sequence, and each one is followed by the code associated with it. Read down the list of decisions until one is found that is satisfied. The code associated with this condition is executed, and then the entire structure is finished. The trailing **else** part handles the "default" case, where none of the other conditions apply. If there is no default action, this final **else** part is omitted:

```
if (x < 0)
      x = 0
else if (x > 100)
      x = 100
```

if-else ambiguity

There is one thing to notice about complicated structures involving nested **if**'s and **else**'s. Consider

```
if (x > 0)
        if (y > 0)
                write(6, 1) x, y
        else
                write(6, 2) y
```

There are two **if**'s and only one **else**. Which **if** does the **else** go with?

This is a genuine ambiguity in Ratfor, as it is in many other programming languages. The ambiguity is resolved in Ratfor (as elsewhere) by saying that in such cases the **else** goes with the closest previous un-else'ed **if**. Thus in this case, the **else** goes with the inner **if**, as we have indicated by the indentation.

It is a wise practice to resolve such cases by explicit braces, just to make your intent clear. In the case above, we would write

```
if (x > 0) {
        if (y > 0)
                write(6, 1) x, y
        else
                write(6, 2) y
}
```

which does not change the meaning, but leaves no doubt in the reader's mind. If we want the other association, we *must* write

```
if (x > 0) {
        if (y > 0)
                write(6, 1) x, y
}
else
        write(6, 2) y
```

The "switch" Statement

The **switch** statement provides a clean way to express multi-way branches which branch on the value of some integer-valued expression. The syntax is

switch (*expression*) {

case *expr1* :
statements
case *expr2, expr3* :
statements
...
default:
statements
}

Each **case** is followed by a list of comma-separated integer expressions. The *expression* inside **switch** is compared against the case expressions *expr1, expr2,* and so on in turn until one matches, at which time the statements following that **case** are executed. If no cases match *expression*, and there is a **default** section, the statements with it are done; if there is no **default**, nothing is done. In all situations, as soon as some block of statements is executed, the entire **switch** is exited immediately. (Readers familiar with C[4] should beware that this behavior is not the same as the C **switch**.)

The "do" Statement

The **do** statement in Ratfor is quite similar to the DO statement in Fortran, except that it uses no statement number. The statement number, after all, serves only to mark the end of the DO, and this can be done just as easily with braces. Thus

```
do i = 1, n {
        x(i) = 0.0
        y(i) = 0.0
        z(i) = 0.0
}
```

is the same as

```
do 10 i = 1, n
        x(i) = 0.0
        y(i) = 0.0
        z(i) = 0.0
10      continue
```

The syntax is:

do *legal-Fortran-DO-text*
Ratfor statement

The part that follows the keyword **do** has to be something that can legally go into a Fortran DO statement. Thus if a local version of Fortran allows DO limits to be expressions (which is not currently permitted in ANSI Fortran), they can be used in a Ratfor **do**.

The *Ratfor statement* part will often be enclosed in braces, but as with the **if**, a single statement need not have braces around it. This code sets an array to zero:

```
do i = 1, n
        x(i) = 0.0
```

Slightly more complicated,

```
do i = 1, n
        do j = 1, n
                m(i, j) = 0
```

sets the entire array **m** to zero, and

```
do i = 1, n
        do j = 1, n
                if (i < j)
                        m(i, j) = -1
                else if (i == j)
                        m(i, j) = 0
                else
                        m(i, j) = +1
```

sets the upper triangle of **m** to −1, the diagonal to zero, and the lower triangle to +1. (The operator == is "equals", that is, ".EQ.".) In each case, the statement that follows the **do** is logically a *single* statement, even though complicated, and thus needs no braces.

"break" and "next"

Ratfor provides a statement for leaving a loop early, and one for beginning the next iteration. **break** causes an immediate exit from the **do**; in effect it is a branch to the statement *after* the **do**. **next** is a branch to the bottom of the loop, so it causes the next iteration to be done. For example, this code skips over negative values in an array:

```
do i = 1, n {
        if (x(i) < 0.0)
                next
        process positive element
}
```

break and **next** also work in the other Ratfor looping constructions that we will talk about in the next few sections.

break and **next** can be followed by an integer to indicate breaking or iterating that level of enclosing loop; thus

```
break 2
```

exits from two levels of enclosing loops, and **break 1** is equivalent to **break**. **next 2** iterates the second enclosing loop. (Realistically, multi-level **break**'s and **next**'s are not likely to be much used because they lead to code that is hard to understand and somewhat risky to change.)

The "while" Statement

One of the problems with the Fortran DO statement is that it generally insists upon being done once, regardless of its limits. If a loop begins

```
DO I = 2, 1
```

this will typically be done once with **I** set to 2, even though common sense would suggest that perhaps it shouldn't be. Of course a Ratfor **do** can easily be preceded by a test

```
if (j <= k)
        do i = j, k {
                ---
        }
```

but this has to be a conscious act, and is often overlooked by programmers.

A more serious problem with the DO statement is that it encourages that a program be written in terms of an arithmetic progression with small positive steps, even though that may not be the best way to write it. If code has to be contorted to fit the requirements imposed by the Fortran DO, it is that much harder to write and understand.

To overcome these difficulties, Ratfor provides a **while** statement, which is simply a loop: "while some condition is true, repeat this group of statements". It has no preconceptions about why one is looping. For example, this routine to compute sin(x) by the Maclaurin series combines two termination criteria.

```
real function sin(x, e)
        # returns sin(x) to accuracy e, by
        # sin(x) = x - x**3/3! + x**5/5! - ...

        sin = x
        term = x

        i = 3
        while (abs(term)>e & i<100) {
                term = -term * x**2 / float(i*(i-1))
                sin = sin + term
                i = i + 2
        }

        return
        end
```

Notice that if the routine is entered with **term** already smaller than **e**, the loop will be done *zero times*, that is, no attempt will be made to compute **x******3** and thus a potential underflow is avoided. Since the test is made at the top of a **while** loop instead of the bottom, a special case disappears — the code works at one of its boundaries. (The test **i<100** is the other boundary — making sure the routine stops after some maximum number of iterations.)

As an aside, a sharp character "#" in a line marks the beginning of a comment; the rest of the line is comment. Comments and code can co-exist on the same line — one can make marginal remarks, which is not possible with Fortran's "C in column 1" convention. Blank lines are also permitted anywhere (they are not in Fortran); they should be used to emphasize the natural divisions of a program.

The syntax of the **while** statement is

```
while (legal Fortran condition)
        Ratfor statement
```

As with the **if**, *legal Fortran condition* is something that can go into a Fortran Logical IF, and *Ratfor statement* is a single statement, which may be multiple statements in braces.

The **while** encourages a style of coding not normally practiced by Fortran programmers. For example, suppose **nextch** is a function which returns the next input character both as a function value and in its argument. Then a loop to find the first non-blank character is just

```
while (nextch(ich) == iblank)
        ;
```

A semicolon by itself is a null statement, which is necessary here to mark the end of the **while**; if it were not present, the **while** would control the next statement. When the loop is broken, **ich** contains the first non-blank. Of course the same code can be written in Fortran as

```
100     if (nextch(ich) .eq. iblank) goto 100
```

but many Fortran programmers (and a few compilers) believe this line is illegal. The language at one's disposal strongly influences how one thinks about a problem.

The "for" Statement

The **for** statement is another Ratfor loop, which attempts to carry the separation of loop-body from reason-for-looping a step further than the **while.** A **for** statement allows explicit initialization and increment steps as part of the statement. For example, a DO loop is just

```
for (i = 1; i <= n; i = i + 1) ...
```

This is equivalent to

```
i = 1
while (i <= n) {
        ...
        i = i + 1
}
```

The initialization and increment of **i** have been moved into the **for** statement, making it easier to see at a glance what controls the loop.

The **for** and **while** versions have the advantage that they will be done zero times if **n** is less than 1; this is not true of the **do.**

The loop of the sine routine in the previous section can be re-written with a **for** as

```
for (i=3; abs(term) > e & i < 100; i=i+2) {
        term = -term * x**2 / float(i*(i-1))
        sin = sin + term
}
```

The syntax of the **for** statement is

```
for ( init ; condition ; increment )
        Ratfor statement
```

init is any single Fortran statement, which gets done once before the loop begins. *increment* is any single Fortran statement, which gets done at the end of each pass through the loop, before the test. *condition* is again anything that is legal in a logical IF. Any of *init, condition,* and *increment* may be omitted, although the semicolons *must* always be present. A non-existent *condition* is treated as always true, so **for(;;)** is an indefinite repeat. (But see the **repeat-until** in the next section.)

The **for** statement is particularly useful for backward loops, chaining along lists, loops that might be done zero times, and similar things which are hard to express with a DO statement, and obscure to write out with IF's and GOTO's. For example, here is a backwards DO loop to find the last non-blank character on a card:

```
for (i = 80; i > 0; i = i - 1)
        if (card(i) != blank)
                break
```

("!=" is the same as ".NE."). The code scans the columns from 80 through to 1. If a non-blank is found, the loop is immediately broken. (**break** and **next** work in **for**'s and **while**'s just as in **do**'s). If **i** reaches zero, the card is all blank.

This code is rather nasty to write with a regular Fortran DO, since the loop must go forward, and we must explicitly set up proper conditions when we fall out of the loop. (Forgetting this is a common error.) Thus:

```
      DO 10 J = 1, 80
          I = 81 - J
          IF (CARD(I) .NE. BLANK) GO TO 11
10    CONTINUE
      I = 0
11    ...
```

The version that uses the **for** handles the termination condition properly for free; **i** *is* zero when we fall out of the **for** loop.

The increment in a **for** need not be an arithmetic progression; the following program walks along a list (stored in an integer array **ptr**) until a zero pointer is found, adding up elements from a parallel array of values:

```
sum = 0.0
for (i = first; i > 0; i = ptr(i))
        sum = sum + value(i)
```

Notice that the code works correctly if the list is empty. Again, placing the test at the top of a loop instead of the bottom eliminates a potential boundary error.

The "repeat-until" statement

In spite of the dire warnings, there are times when one really needs a loop that tests at the bottom after one pass through. This service is provided by the **repeat-until**:

repeat
Ratfor statement
until (*legal Fortran condition*)

The *Ratfor statement* part is done once, then the condition is evaluated. If it is true, the loop is exited; if it is false, another pass is made.

The **until** part is optional, so a bare **repeat** is the cleanest way to specify an infinite loop. Of course such a loop must ultimately be broken by some transfer of control such as **stop**, **return**, or **break**, or an implicit stop such as running out of input with a READ statement.

As a matter of observed fact[8], the **repeat-until** statement is *much* less used than the other looping constructions; in particular, it is typically outnumbered ten to one by **for** and **while**. Be cautious about using it, for loops that test only at the bottom often don't handle null cases well.

More on break and next

break exits immediately from **do**, **while**, **for**, and **repeat-until**. **next** goes to the test part of **do**, **while** and **repeat-until**, and to the increment step of a **for**.

"return" Statement

The standard Fortran mechanism for returning a value from a function uses the name of the function as a variable which can be assigned to; the last value stored in it is the function value upon return. For example, here is a routine **equal** which returns 1 if two arrays are identical, and zero if they differ. The array ends are marked by the special value −1.

```
# equal _ compare str1 to str2;
#     return 1 if equal, 0 if not
      integer function equal(str1, str2)
      integer str1(100), str2(100)
      integer i

      for (i = 1; str1(i) == str2(i); i = i + 1)
          if (str1(i) == -1) {
              equal = 1
              return
          }
      equal = 0
      return
      end
```

In many languages (e.g., PL/I) one instead says

return (*expression*)

to return a value from a function. Since this is often clearer, Ratfor provides such a **return** statement — in a function F, **return**(expression) is equivalent to

{ F = expression; return }

For example, here is **equal** again:

```
# equal _ compare str1 to str2;
#     return 1 if equal, 0 if not
      integer function equal(str1, str2)
      integer str1(100), str2(100)
      integer i

      for (i = 1; str1(i) == str2(i); i = i + 1)
          if (str1(i) == -1)
              return(1)
      return(0)
      end
```

If there is no parenthesized expression after **return**, a normal RETURN is made. (Another version of **equal** is presented shortly.)

Cosmetics

As we said above, the visual appearance of a language has a substantial effect on how easy it is to read and understand programs. Accordingly, Ratfor provides a number of cosmetic facilities which may be used to make programs more readable.

Free-form Input

Statements can be placed anywhere on a line; long statements are continued automatically, as are long conditions in **if**, **while**, **for**, and **until**. Blank lines are ignored. Multiple statements may appear on one line, if they are separated by semicolons. No semicolon is needed at the end of a line, if Ratfor can make

some reasonable guess about whether the statement ends there. Lines ending with any of the characters

= + - * , | & (_

are assumed to be continued on the next line. Underscores are discarded wherever they occur; all others remain as part of the statement.

Any statement that begins with an all-numeric field is assumed to be a Fortran label, and placed in columns 1-5 upon output. Thus

```
write(6, 100); 100 format("hello")
```

is converted into

```
      write(6, 100)
100   format(5hhello)
```

Translation Services

Text enclosed in matching single or double quotes is converted to **nH...** but is otherwise unaltered (except for formatting — it may get split across card boundaries during the reformatting process). Within quoted strings, the backslash '\' serves as an escape character: the next character is taken literally. This provides a way to get quotes (and of course the backslash itself) into quoted strings:

```
"\\\'"
```

is a string containing a backslash and an apostrophe. (This is *not* the standard convention of doubled quotes, but it is easier to use and more general.)

Any line that begins with the character '%' is left absolutely unaltered except for stripping off the '%' and moving the line one position to the left. This is useful for inserting control cards, and other things that should not be transmogrified (like an existing Fortran program). Use '%' only for ordinary statements, not for the condition parts of **if**, **while**, etc., or the output may come out in an unexpected place.

The following character translations are made, except within single or double quotes or on a line beginning with a '%'.

==	.eq.	!=	.ne.
>	.gt.	>=	.ge.
<	.lt.	<=	.le.
&	.and.	|	.or.
!	.not.	¬	.not.

In addition, the following translations are provided for input devices with restricted character sets.

[	{	]	}
$(	{	$)	}

"define" Statement

Any string of alphanumeric characters can be defined as a name; thereafter, whenever that name occurs in the input (delimited by non-alphanumerics) it is replaced by the rest of the definition line. (Comments and trailing white spaces are stripped off). A defined name can be arbitrarily long, and must begin with a letter.

define is typically used to create symbolic parameters:

```
define ROWS 100
define COLS 50

dimension a(ROWS), b(ROWS, COLS)

      if (i > ROWS | j > COLS) ...
```

Alternately, definitions may be written as

```
define(ROWS, 100)
```

In this case, the defining text is everything after the comma up to the balancing right parenthesis; this allows multi-line definitions.

It is generally a wise practice to use symbolic parameters for most constants, to help make clear the function of what would otherwise be mysterious numbers. As an example, here is the routine **equal** again, this time with symbolic constants.

```
define    YES     1
define    NO      0
define    EOS     -1
define    ARB     100

# equal _ compare str1 to str2;
#     return YES if equal, NO if not
      integer function equal(str1, str2)
      integer str1(ARB), str2(ARB)
      integer i

      for (i = 1; str1(i) == str2(i); i = i + 1)
            if (str1(i) == EOS)
                  return(YES)
      return(NO)
      end
```

"include" Statement

The statement

```
include file
```

inserts the file found on input stream *file* into the Ratfor input in place of the **include** statement. The standard usage is to place COMMON blocks on a file, and **include** that file whenever a copy is needed:

```
subroutine x
        include commonblocks
        ...
        end

suroutine y
        include commonblocks
        ...
        end
```

This ensures that all copies of the COMMON blocks are identical

Pitfalls, Botches, Blemishes and other Failings

Ratfor catches certain syntax errors, such as missing braces, **else** clauses without an **if**, and most errors involving missing parentheses in statements. Beyond that, since Ratfor knows no Fortran, any errors you make will be reported by the Fortran compiler, so you will from time to time have to relate a Fortran diagnostic back to the Ratfor source.

Keywords are reserved — using **if**, **else**, etc., as variable names will typically wreak havoc. Don't leave spaces in keywords. Don't use the Arithmetic IF.

The Fortran **nH** convention is not recognized anywhere by Ratfor; use quotes instead.

3. IMPLEMENTATION

Ratfor was originally written in C[4] on the UNIX operating system[5]. The language is specified by a context free grammar and the compiler constructed using the YACC compiler-compiler[6].

The Ratfor grammar is simple and straightforward, being essentially

```
prog    : stat
        | prog  stat
stat    : if (...) stat
        | if (...) stat else stat
        | while (...) stat
        | for (...; ...; ...) stat
        | do ... stat
        | repeat stat
        | repeat stat until (...)
        | switch (...) { case ...: prog ...
                         default: prog }
        | return
        | break
        | next
        | digits  stat
        | { prog }
        | anything unrecognizable
```

The observation that Ratfor knows no Fortran follows directly from the rule that says a statement is "anything unrecognizable". In fact most of Fortran falls into this category, since any statement that does not begin with one of the keywords is by definition "unrecognizable."

Code generation is also simple. If the first thing on a source line is not a keyword (like **if**, **else**, etc.) the entire statement is simply copied to the output with appropriate character translation and formatting. (Leading digits are treated as a label.) Keywords cause only slightly more complicated actions. For example, when **if** is recognized, two consecutive labels L and L+1 are generated and the value of L is stacked. The condition is then isolated, and the code

```
if (.not. (condition)) goto L
```

is output. The *statement* part of the **if** is then translated. When the end of the statement is encountered (which may be some distance away and include nested **if**'s, of course), the code

```
L       continue
```

is generated, unless there is an **else** clause, in which case the code is

```
        goto L+1
L       continue
```

In this latter case, the code

```
L+1     continue
```

is produced after the *statement* part of the **else**. Code generation for the various loops is equally simple.

One might argue that more care should be taken in code generation. For example, if there is no trailing **else**,

```
if (i > 0) x = a
```

should be left alone, not converted into

```
        if (.not. (i .gt. 0)) goto 100
        x = a
100     continue
```

But what are optimizing compilers for, if not to improve code? It is a rare program indeed where this kind of "inefficiency" will make even a measurable difference. In the few cases where it is important, the offending lines can be protected by '%'.

The use of a compiler-compiler is definitely the preferred method of software development. The language is well-defined, with few syntactic irregularities. Implementation is quite simple; the original construction took under a week. The language is sufficiently simple, however, that an *ad hoc* recognizer can be readily constructed to do the same job if no compiler-compiler is available.

The C version of Ratfor is used on UNIX and on the Honeywell GCOS systems. C compilers are not as widely available as Fortran, however, so there is also a Ratfor written in itself and originally bootstrapped with the C version. The Ratfor version was written so as to translate into the portable subset of Fortran described in [1], so it is portable, having been run essentially without change on at least twelve distinct machines. (The main restrictions of the portable subset are: only one character per machine word; subscripts in the form $c*v\pm c$; avoiding expressions in places like DO loops; consistency in subroutine argument usage, and in COMMON declarations. Ratfor itself will not gratuitously generate non-standard Fortran.)

The Ratfor version is about 1500 lines of Ratfor (compared to about 1000 lines of C); this compiles into 2500 lines of Fortran. This expansion ratio is somewhat higher than average, since the compiled code contains unnecessary occurrences of COMMON declarations. The execution time of the Ratfor version is dominated by two routines that read and write cards. Clearly these routines could be replaced by machine coded local versions; unless this is done, the efficiency of other parts of the translation process is largely irrelevant.

4. EXPERIENCE

Good Things

"It's so much better than Fortran" is the most common response of users when asked how well Ratfor meets their needs. Although cynics might consider this to be vacuous, it does seem to be true that decent control flow and cosmetics converts Fortran from a bad language into quite a reasonable one, assuming that Fortran data structures are adequate for the task at hand.

Although there are no quantitative results, users feel that coding in Ratfor is at least twice as fast as in Fortran. More important, debugging and subsequent revision are much faster than in Fortran. Partly this is simply because the code can be *read*. The looping statements which test at the top instead of the bottom seem to eliminate or at least reduce the occurrence of a wide class of boundary errors. And of course it is easy to do structured programming in Ratfor; this self-discipline also contributes markedly to reliability.

One interesting and encouraging fact is that programs written in Ratfor tend to be as readable as programs written in more modern languages like Pascal. Once one is freed from the shackles of Fortran's clerical detail and rigid input format, it is easy to write code that is readable, even esthetically pleasing. For example, here is a Ratfor implementation of the linear table search discussed by Knuth [7]:

```
A(m+1) = x
for (i = 1; A(i) != x; i = i + 1)
        ;
if (i > m) {
        m = i
        B(i) = 1
}
else
        B(i) = B(i) + 1
```

A large corpus (5400 lines) of Ratfor, including a subset of the Ratfor preprocessor itself, can be found in [8].

Bad Things

The biggest single problem is that many Fortran syntax errors are not detected by Ratfor but by the local Fortran compiler. The compiler then prints a message in terms of the generated Fortran, and in a few cases this may be difficult to relate back to the offending Ratfor line, especially if the implementation conceals the generated Fortran. This problem could be dealt with by tagging each generated line with some indication of the source line that created it, but this is inherently implementation-dependent, so no action has yet been taken. Error message interpretation is actually not so arduous as might be thought. Since Ratfor generates no variables, only a simple pattern of IF's and GOTO's, data-related errors like missing DIMENSION statements are easy to find in the Fortran. Furthermore, there has been a steady improvement in Ratfor's ability to catch trivial syntactic errors like unbalanced parentheses and quotes.

There are a number of implementation weaknesses that are a nuisance, especially to new users. For example, keywords are reserved. This rarely makes any difference, except for those hardy souls who want to use an Arithmetic IF. A few standard Fortran constructions are not accepted by Ratfor, and this is perceived as a problem by users with a large corpus of existing Fortran programs. Protecting every line with a

'%' is not really a complete solution, although it serves as a stop-gap. The best long-term solution is provided by the program Struct [9], which converts arbitrary Fortran programs into Ratfor.

Users who export programs often complain that the generated Fortran is "unreadable" because it is not tastefully formatted and contains extraneous CONTINUE statements. To some extent this can be ameliorated (Ratfor now has an option to copy Ratfor comments into the generated Fortran), but it has always seemed that effort is better spent on the input language than on the output esthetics.

One final problem is partly attributable to success — since Ratfor is relatively easy to modify, there are now several dialects of Ratfor. Fortunately, so far most of the differences are in character set, or in invisible aspects like code generation.

5. CONCLUSIONS

Ratfor demonstrates that with modest effort it is possible to convert Fortran from a bad language into quite a good one. A preprocessor is clearly a useful way to extend or ameliorate the facilities of a base language.

When designing a language, it is important to concentrate on the essential requirement of providing the user with the best language possible for a given effort. One must avoid throwing in "features" — things which the user may trivially construct within the existing framework.

One must also avoid getting sidetracked on irrelevancies. For instance it seems pointless for Ratfor to prepare a neatly formatted listing of either its input or its output. The user is presumably capable of the self-discipline required to prepare neat input that reflects his thoughts. It is much more important that the language provide free-form input so he *can* format it neatly. No one should read the output anyway except in the most dire circumstances.

Acknowledgements

C. A. R. Hoare once said that "One thing [the language designer] should not do is to include untried ideas of his own." Ratfor follows this precept very closely — everything in it has been stolen from someone else. Most of the control flow structures are taken directly from the language C[4] developed by Dennis Ritchie; the comment and continuation conventions are adapted from Altran[10].

I am grateful to Stuart Feldman, whose patient simulation of an innocent user during the early days of Ratfor led to several design improvements and the eradication of bugs. He also translated the C parse-tables and YACC parser into Fortran for the first Ratfor version of Ratfor.

References

[1] B. G. Ryder, "The PFORT Verifier," *Software—Practice & Experience*, October 1974.

[2] American National Standard Fortran. American National Standards Institute, New York, 1966.

[3] *For-word: Fortran Development Newsletter*, August 1975.

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

[5] D. M. Ritchie and K. L. Thompson, "The UNIX Time-sharing System." *CACM*, July 1974.

[6] S. C. Johnson, "YACC — Yet Another Compiler-Compiler." Bell Laboratories Computing Science Technical Report #32, 1978.

[7] D. E. Knuth, "Structured Programming with goto Statements." *Computing Surveys*, December 1974.

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

[9] B. S. Baker, "Struct — A Program which Structures Fortran", Bell Laboratories internal memorandum, December 1975.

[10] A. D. Hall, "The Altran System for Rational Function Manipulation — A Survey." *CACM*, August 1971.

PWB/Graphics Overview

A. R. Feuer

Bell Laboratories
Piscataway, New Jersey 08854

1. INTRODUCTION

PWB/Graphics, or just *graphics*, is the name given to a growing collection of numerical and graphical commands available as part of the Programmer's Workbench [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. $y=x^2, 0 \leqslant x \leqslant 10$. 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 $y=x^2, 0 \leqslant x \leqslant 10$ and display it on a Tektronix* display terminal would be:

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

- **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.

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.

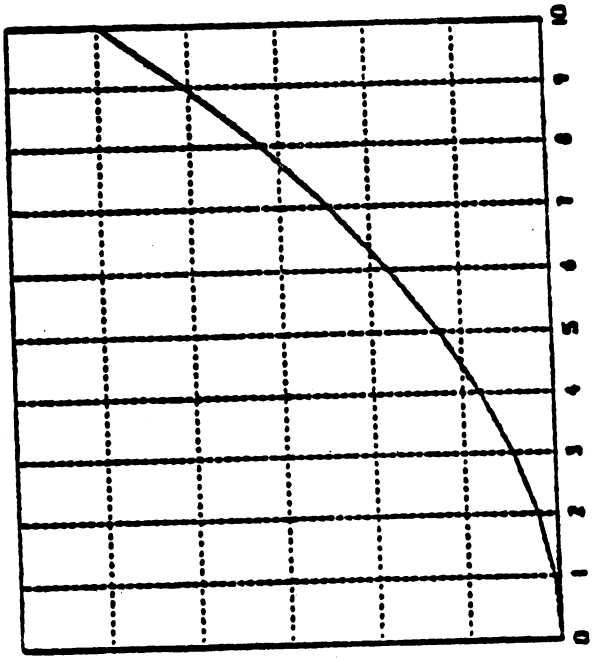


Figure 2. gas -s0.t10 | sf "x^2" | plot -x11,ya | cd

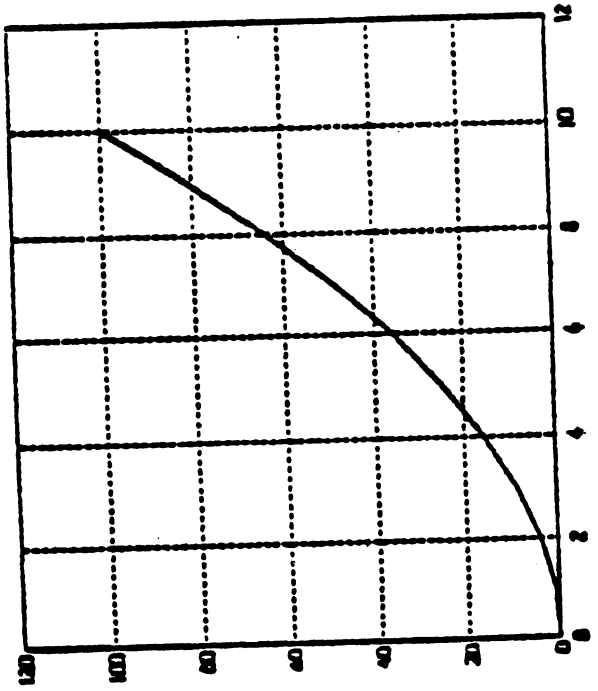


Figure 1. gas -s0.t10 | sf "x^2" | plot | cd

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 PWB/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. *Plot*(1) filters can be used in conjunction with *gtop* (see *gutil*(1)) 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 logoff UNIX from *graphics* type **quit**.

4. EXAMPLES OF WHAT YOU CAN DO

4.1 Numerical Manipulation and Plotting

Stat(1) 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,l10 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 filenames. If a filename 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.

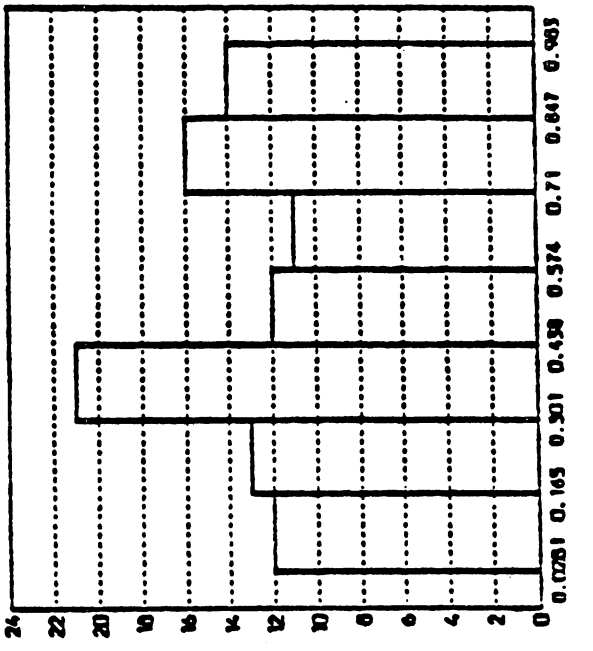


Figure 4. Histogram of 100 random numbers

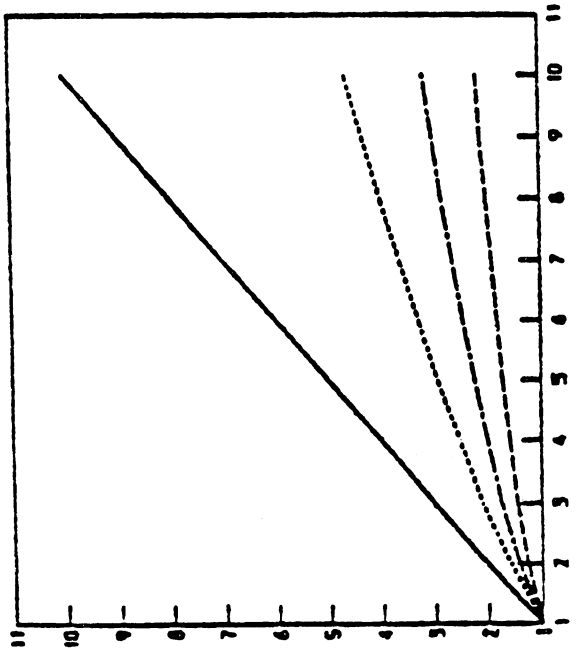


Figure 3. Some roots of the first ten integers

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 PWB/UNIX system and use it. Tutorials exist for *stat*(1) and *ged*(1). [2] contains administrative information for *graphics*. Reference information can be found in the PWB/UNIX User's Manual under the following manual pages:

ged(1), the graphical editor;
gps(5), a description of a graphical primitive string;
graphics(1), the entry point for *graphics*;
gutil(1), a collection of utility commands;
stat(1), numerical manipulation and plotting commands;
tek4000(1), a collection of commands to manipulate Tektronix 4000 series terminals; and
toc(1), routines to build tables of contents.

6. REFERENCES

[1] *PWB/UNIX User's Manual* – Release 2.0., Bell Laboratories, 1979.

[2] R. L. Chen and D. E. Pinkston, *Administrative Information for PWB/Graphics*, Bell Laboratories Memorandum, 1979.

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.    cvrtcpt.d       7
2.2.    gtop.d          8
2.3.    ptop.d          9
3.      stat.d          4
4.      tek4000.d       5
4.1.    gsd.d           37
4.4.    td.d            6
5.      tioc.d          3
5.1.    ttioc.d         3
5.2.    vtioc.d         22
6.      whatis.d        108
```

Figure 5a. Dtoc output

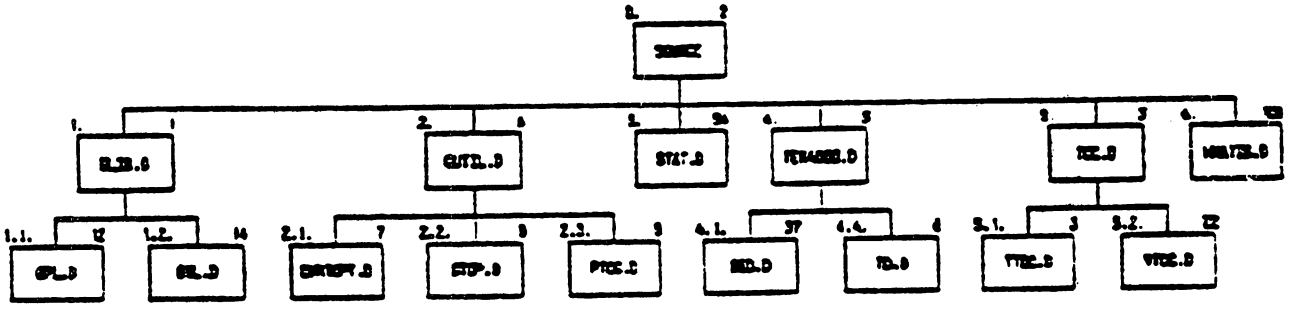


Figure 5b. Vtoc output

Administrative Information For PWB/Graphics

Ruth L. Chen
Diane E. Pinkston

Bell Laboratories
Piscataway, New Jersey 08854

1. INTRODUCTION

This document is a reference guide for system administrators who are using or establishing a PWB/Graphics facility [1] on UNIX™. It contains information about directory structure, installation, makefiles, hardware requirements, and miscellaneous facilities of PWB/Graphics.

2. PWB/Graphics STRUCTURE

Figure 1 contains a graphical representation of the directory structure of PWB/Graphics. In this paper, the *Shell* variable $SRC will represent the parent node for graphics source. On PWB/UNIX $SRC is /usr/src/cmd. If PWB/Graphics is copied onto other systems, $SRC could have other values but should, in general, be the same as on PWB/UNIX.

The *graphics* command (see *graphics*(1)) resides in /usr/bin. All other PWB/Graphics executables are located in /usr/bin/graf. /usr/lib/graf contains text for whatis documentation (see *gutil*(1)) and editor scripts for *toc* (see *toc*(1)).

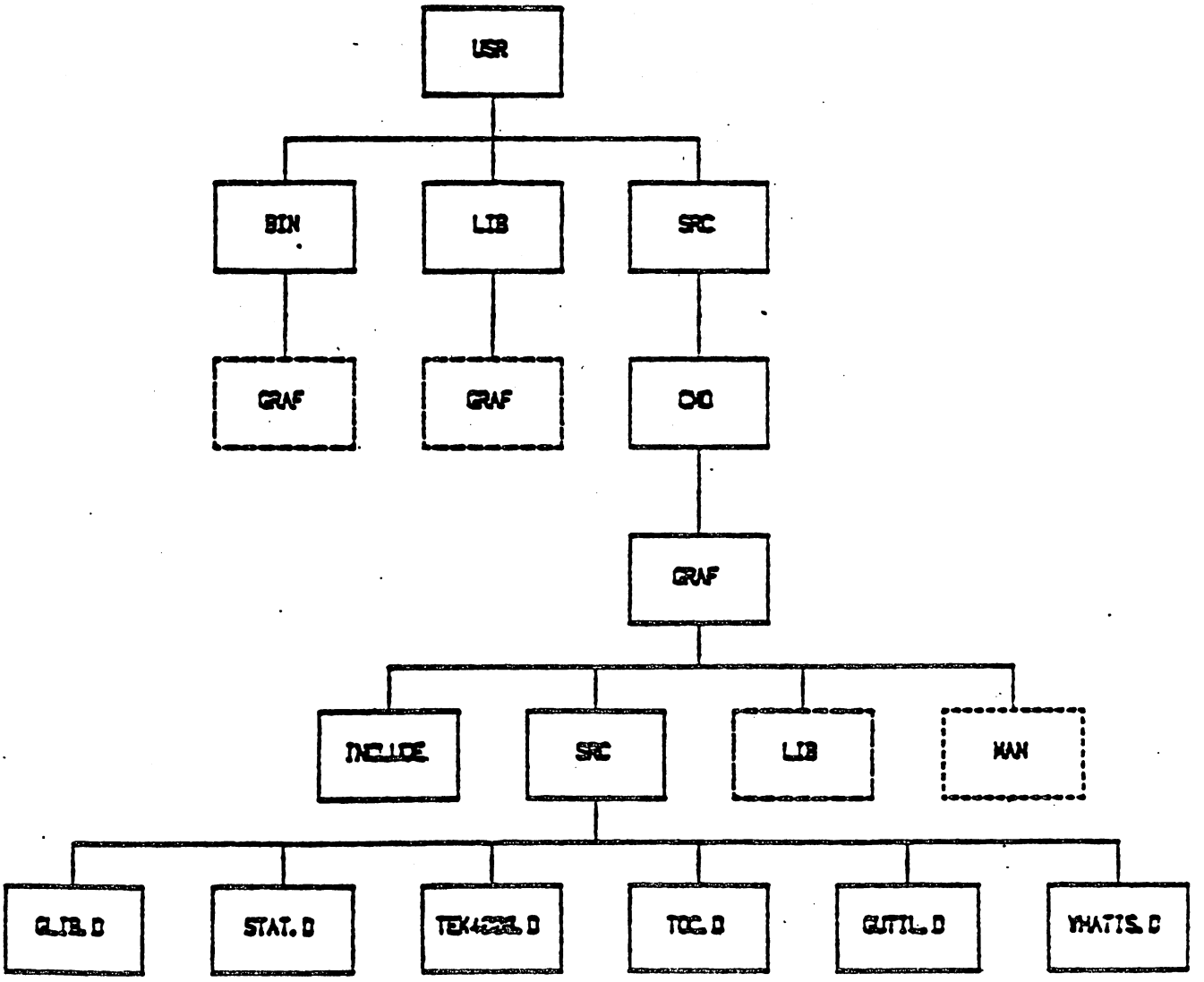
PWB/Graphics source resides below the directory $SRC/graf. $SRC/graf is broken into the following subdirectories:

- 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 in $SRC/graf/lib/glib.a.
  - stat.d - contains source for numerical analysis and plotting routines.
  - tek4000.d - contains source for *ged* (the graphical editor), *id* (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 mm files and the install routine for quick-reference documentation.
- lib - contains glib.a which contains commonly used graphical subroutines.
- man - Figure 1 shows $SRC/graf/man as a dotted box because this directory does not exist on PWB/UNIX systems where all manual pages reside in /usr/man. $SRC/graf/man is created if PWB/Graphics is copied onto another system (see section 3.), and will contain the following manual page files: graphics.1, gutil.1, stat.1, tek4000.1, toc.1, ged.1 and gps.5.

3. INSTALLING PWB/Graphics

Procedures for installing PWB/Graphics:

Fig. 1 PWB/Graphics Structure



1. PWB/UNIX systems,
   - To build the entire Graphics system (i.e. all boxes except man in Figure 1), execute (as superuser)

     ./:mkcmd graf

     ./:mkcmd resides in /usr/src, and all manual pages exist in /usr/man.
   - To build a particular subsystem, execute

     ./:mkcmd graf *subsystem*
   - To build a particular *command* within a subsystem, execute

     ./:mkcmd graf *subsystem command-name*
2. UNIX/TS systems not running PWB,
   - See appendix for tape copying procedures.
   - Build $SRC/graf/lib and PWB/Graphics executables (dashed boxes in Figure 1) by typing:

     make -f $SRC/graf/graf.mk

   - To make a particular graphics subsystem use the Makefile in $SRC/graf/src, e.g.

     cd $SRC/graf/src
     make *subsystem*
   - Note, there is a name conflict between PWB/Graphics *plot* and UNIX/TS *plot*(1). The recommended fix is to remove /usr/bin/plot and move the *plot*(1) filters from /usr/lib to /usr/bin.

A *subsystem* is either *glib*, *stat*, *tek4000*, *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/[nt]pwbmm.m*	Programmer's Workbench memorandum macros for *[nt]roff*, used by the whatis subsystem.

The build process takes approximately one hour of system time. If the make must be stopped, it is a good idea to rebuild from the top 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 PWB/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 the Makefile in $SRC/graf/src.

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 parenthesis and an explanation of their usage.

1. $SRC/graf/graf.mk

   BIN1 (/usr/bin) installation directory for the *graphics* command.

   BIN2 (/usr/bin/graf) installation directory for other graphic commands.

   SRC (/usr/src/cmd) parent directory for source code.

2. $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.

3. $SRC/graf/src/stat.d/Makefile

   BIN (../../bin) installation directory for executable commands.

4. $SRC/graf/src/toc.d/Makefile

   BIN (../../bin) installation directory for executable commands.

5. $SRC/graf/src/tek4000.d/Makefile

   BIN (../../bin) installation directory for executable commands.

6. $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*, in /a1/pmt/dp/bin:

```
cd $SRC/graf/src/tek4000.d
make BIN=/a1/pmt/dp/bin ged
```

4. TEKTRONIX TERMINAL

The PWB/Graphics display function *id* and the graphical editor *ged* both use Tektronix Series 4010 storage tubes. Below is a list of device considerations necessary for PWB/Graphics operation.

4.1 Getty Table Entry

When a Tektronix 4010 series terminal is connected via a dedicated line to UNIX, an entry in the system table (in /usr/src/cmd/getty.c) is suggested, to store terminal status information. This table entry appears as follows on PWB/UNIX:

```
/* table '6' -- 4800/9600 -- tektronix 4014 */
        '6', 7,
        ANYP+ RAW+ FF1, ANYP+ ECHO+ CRMOD+ FF1,
        B4800, B4800,
        "\033\014\000login: ",

        7, '6',
        ANYP+ RAW+ FF1, ANYP+ ECHO+ CRMOD+ FF1,
        B9600, B9600,
        "\033\014\000login: ",
```

but on other systems it may have to be created and then referenced in /etc/inittab. Standard parity and a form-feed delay are necessary. The form-feed delay gives the screen time to clear without losing information. Below is an example of the terminal status as printed by *stty*:

```
speed 4800 baud
erase = '#'; kill = '@'
even odd -nl echo -tabs ff1
```

4.2 Strap Options

The standard strap options as listed below should be used (see the Reference Manual for the Tektronix 4014 [3]):

1. LF effect - LF causes line-feed only.
2. CR effect - CR caused carriage return only.
3. Del implies loy - Del key is interpreted as low-order y value.
4. Graphics Input terminators - None.

4.3 Enhanced Graphics Module

The Enhanced Graphics Module of Tektronix terminals is required for PWB/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).

5. MISCELLANEOUS INFORMATION

5.1 Announcements

The *graphics* command provides a means of printing out announcements to users. To set up an anouncement 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 pathname of the announce file.

5.2 Uselog

The *graphics* command also provides a means of monitoring its use by listing users in a file. To set up a uselog facility create a writeable 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**.

5.3 Restricted Environments

Restricted environments can be used to limit user access to the system (see *rsh*(1) [4]). In a restricted environment, commands in /rbin and /usr/rbin are executed before those in /bin and /usr/bin. The commands *ed*, *mv*, *rm*, and *sh* require restricted interface programs which do not allow users to move or remove files that begin with dot (.)[2].

Creating restricted environments for graphics:

1. Create a restricted *ged* command in /usr/rbin as follows:

```
exec /usr/bin/graf/ged -R
```

2. 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.
3. Call graphics -r from .profile.

The execution of graphics -r changes $PATH to look for commands in /rbin and /usr/rbin 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 Alan 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 PWB/UNIX environment.

REFERENCES

[1] Feuer, A. R. *PWB/Graphics Overview*. Bell Laboratories, 1979.

[2] Petrella, M. J. *Restricted Access to PWB/UNIX* – DRAFT. Bell laboratories, May 1979.

[3] Tektronix. *User's Manual for 4014 and 4014-1 Display Terminal*. July, 1974.

[4] *PWB/UNIX User's Manual* – Release 2.0.

APPENDIX

Procedures for tape copying (as superuser)

- Locate graphics source by changing directory to $SRC, the parent directory.
- Then copy source and manual pages from the tape by typing

```
cpio -idm < /dev/mt4        (creates graf)
cd graf
cpio -idm < /dev/mt0        (creates man)
```

This will result in the directory structure indicated by the solid boxes plus $SRC/graf/man in Figure 1. Necessary sub-directories will be created (see *cpio*(1)).

January 1980

A Tutorial Introduction to the Graphical Editor

Alan R. Feuer

Bell Laboratories
Piscataway, New Jersey 08854

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 PWB/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, **<return>** means return and **<sp>** means space. In the examples, output from the terminal is printed in normalface type. Inline comments are in normalface and are surrounded by parentheses.

2. COMMANDS

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

ged <return>

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 **<return>** . The **<return>** 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*] **<return>**

? is an example of a simple command. It lists the commands and options understood by *ged*. Type

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

to generate the list.

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 <return> causes the erase action to occur. Typing <rubout> after a command name and before the final <return> for the command aborts the command. Thus while

•erase <return>

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 <return>

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 <return>

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 <return>

Next enter the *text* (stage 2):

a line of text <return>.

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

<position cursor> <return>

Positioning of the graphic cursor is done either with the thumbwheel knobs on the terminal keyboard or with an auxiliary joystick. The <return> establishes the location of the cursor to be the starting point for the text string. The *Text* command ends at stage 3, so this <return> 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 <return> (i.e., \<return>) to indicate continuation. To illustrate some of these capabilities, try the following:

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

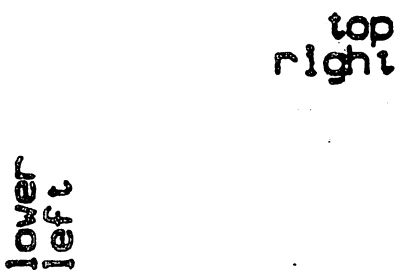


Figure 1. Generating text objects

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* <return>

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 <return>
<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)
<return>                             (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 <return> the points are connected in numerical order.

2.2.2.1 Accessing points by name. 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 of Section 2.2.2 by entering:

```
•Lines <return>
<position cursor> <sp>
<position cursor> <sp>
<position cursor> <sp>
$0 <return>                          (reference point 0)
<return>
```

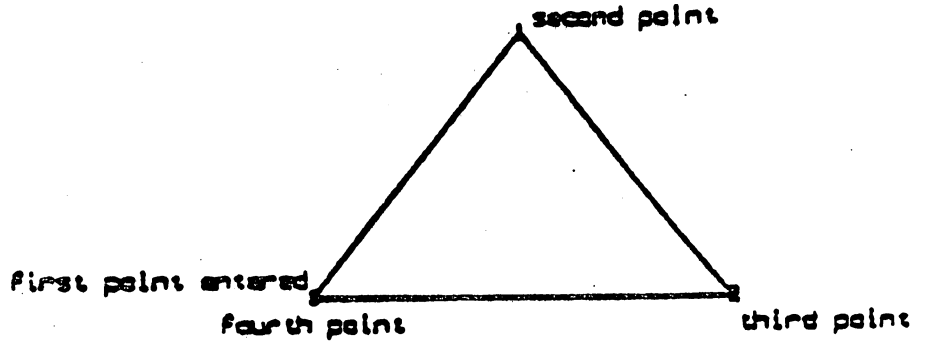


Figure 2. Building a triangle

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

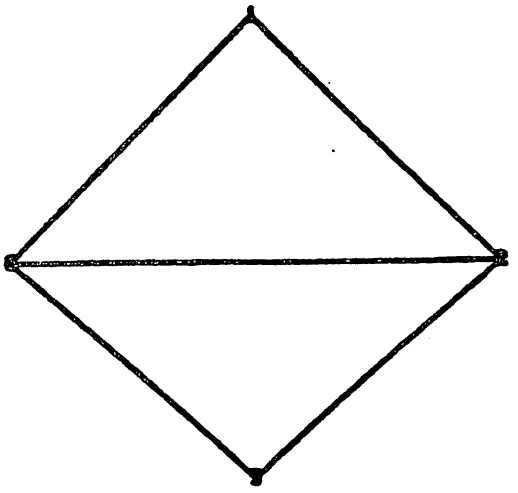


Figure 3. Accessing the previous point set

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

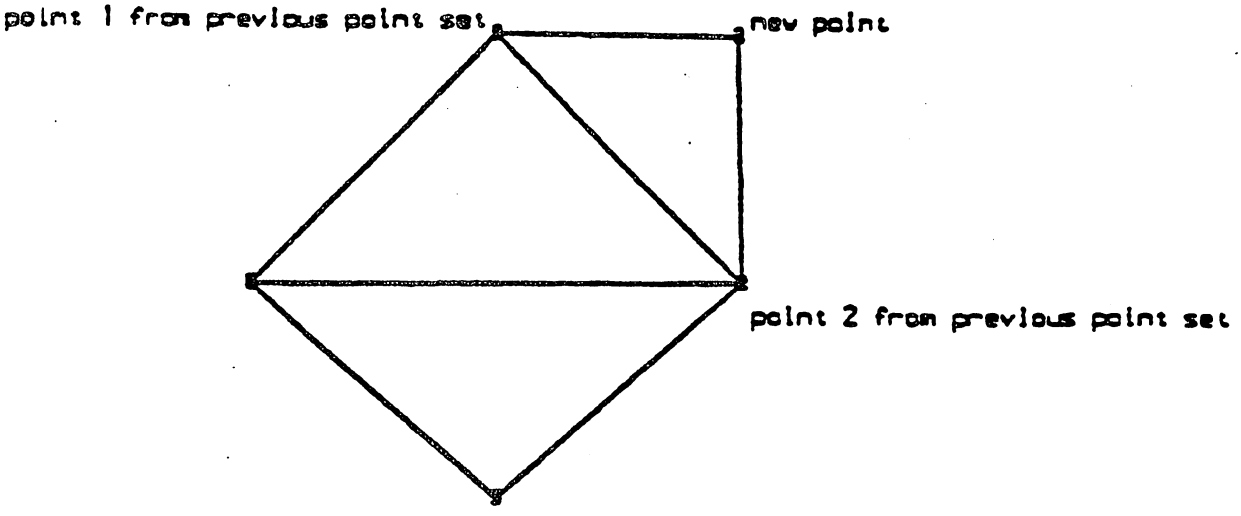


Figure 4. Referencing points from the previous point set

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 <return>
<position cursor> <sp>          (enter a point)
>A                              (name the point A)
<position cursor> <sp>
<return>
```

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

```
•Lines <return>
$A
<position cursor> <sp>
<return>
```

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 <return>
<position cursor> <sp>
<position cursor> <sp>
$0 <return>
<return>
```

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 <return>
```

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

The defined-area is outlined with dotted lines. The reason for the seemingly extra <return> 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 <return> . 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 <return> . Using a zero point defined-area, the *Delete* command would be:

```
•Delete <return>
<position cursor>          (center the crosshairs on the object-handle)
<return>                   (terminate the defined-area)
<return>                   (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 universe 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 <return>
<position cursor> <return>     (centered on the object-handle)
<return>                       (this establishes a pivot, marked with an asterisk)
<position cursor> <return>     (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 <return>
<position cursor> <sp>
<position cursor> <return>
```

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

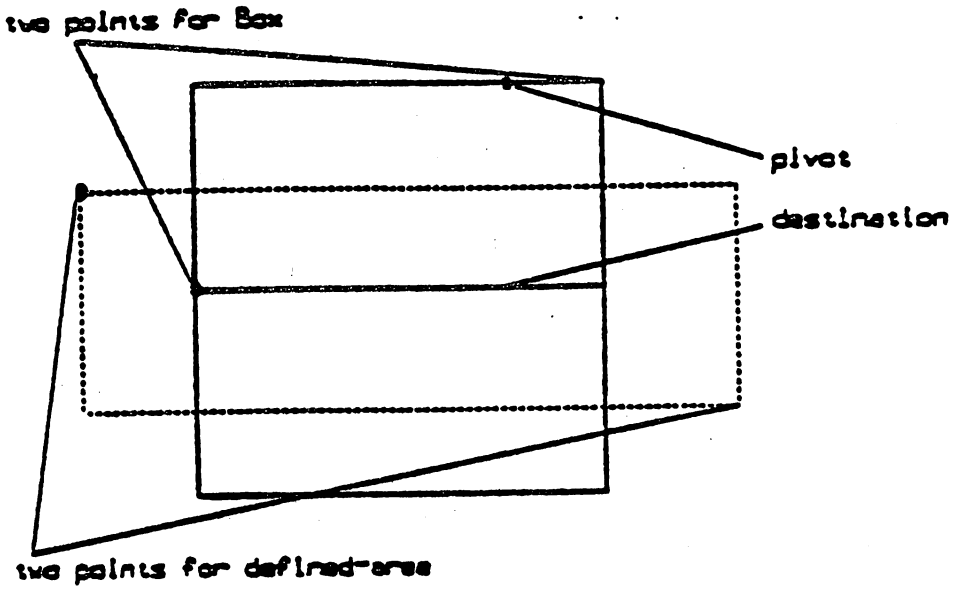


Figure 5. Growing a box

Since 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 <return>	(factor is in percent)
<position cursor> <return>	(point to object-handle)
<position cursor> <return>	(set pivot to rightmost character)
<return>	
•Scale -f50 <return>	
. <return>	(reference the previous defined-area)
<position cursor> <return>	(set pivot above a character near the middle)
<return>	

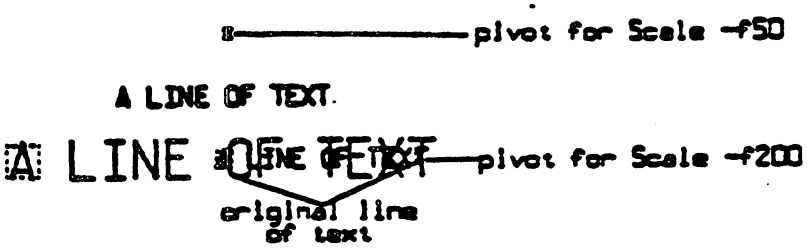


Figure 6. Scaling text

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 <return>
<position cursor> <sp>          (specify the center)
<position cursor> <return>      (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 <return>        (height is in universe units, see Section 2.4)
<position cursor> <return>      (point to the object-handle)
<return>
```

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 <return>           (angle is in degrees)
<position cursor> <return>      (point to object-handle)
<position cursor> <return>      (set pivot to rightmost character)
<return>

•Rotate −a−90 <return>
. <return>                      (reference previous defined-area)
<position cursor> <return>      (set pivot to a character near the middle)
<return>
```

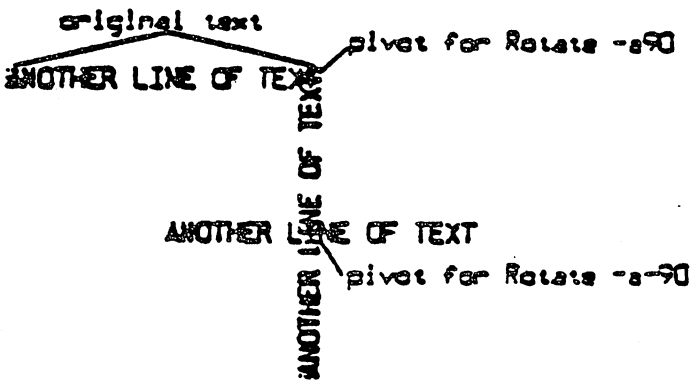


Figure 7. Rotating text

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 (solid, dashed, dot-dashed, dotted, long-dashed) and three widths (narrow, medium, bold). Style is controlled by the s option, width by w.

```
•Lines -wn,sdo <return>
<position cursor> <sp>
<position cursor> <sp>
<return>
```

creates a narrow width dotted line.

```
•Edit -wb,sdd <return>
<position cursor> <return>        (point to object-handle of the line)
<return>
```

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 <return>
```

displays the entire universe.

2.4.1 Windowing. 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 <return>
<position cursor> <sp>          (above and to the left of any object)
<position cursor> <return>      (below and to the right, also end points)
<return>                        (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 out 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 <return>
<position cursor> <return>        (locate a point in the universe)
<position cursor> <return>        (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* <return>

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* <return>

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 <return>
•view −h <return>
```

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* <return>

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 things up. 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 specifed 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 ($0 \leqslant n < 1280$).
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

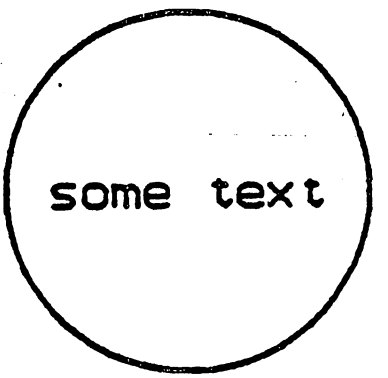
[1] Feuer, A. R.; "PWB/Graphics Overview"; TM 79-3782-1, June 11, 1979.

[2] *PWB/UNIX User's Manual* Release 2.0, Bell Laboratories, 1979.

APPENDIX: SOME EXAMPLES OF WHAT CAN BE DONE

1. Text Centered Within a Circle

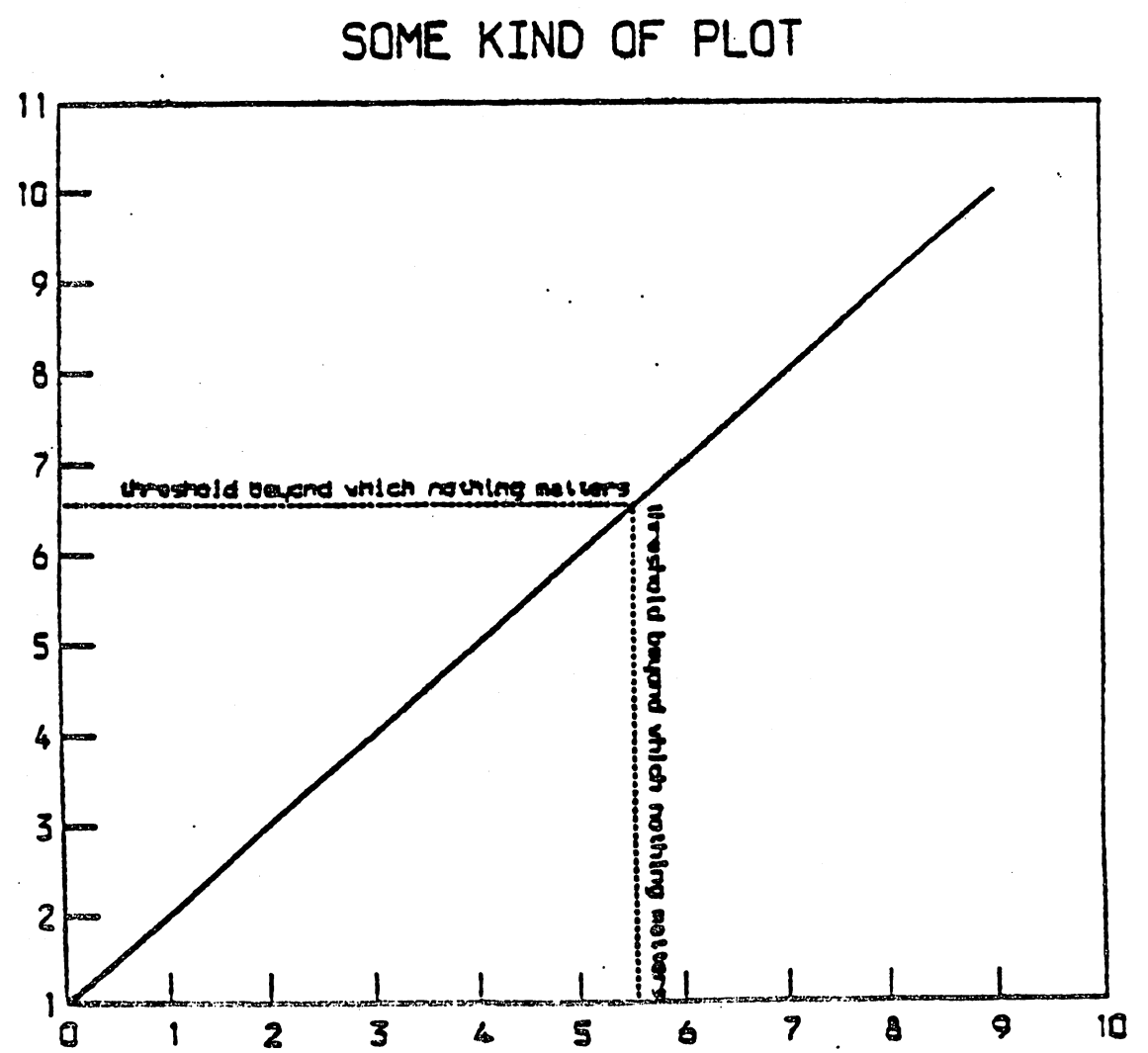
•Circle <cr>	
<position cursor> <sp>	(establish center)
<position cursor> <cr>	(establish radius)
•Text −m <cr>	(text is to be centered)
some text <cr>	
$.0 <cr>	(first point from previous set, i.e., circle center)
<cr>	



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)
```

January 1980

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

Lex - A Lexical Analyzer Generator

M. E. Lesk and E. Schmidt
Bell Laboratories
Murray Hill, New Jersey 07974

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

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

The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it.

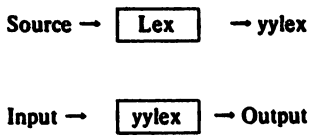
Lex can be used to generate analyzers in either C or Ratfor, a language which can be translated automatically to portable Fortran. It is available on the PDP-11 UNIX, Honeywell GCOS, and IBM OS systems. Lex is designed to simplify interfacing with Yacc, for those with access to this compiler-compiler system.

Table of Contents

1 Introduction.

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to



An overview of Lex

Figure 1

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 there are only two host languages, C[1] and Fortran (in the form of the Ratfor language[2]). Lex itself exists on UNIX, GCOS, and OS/370; but the code generated by Lex may be taken anywhere the appropriate compilers exist.

Lex turns the user's expressions and actions (called *source* in this memo) into the host general-purpose language; the generated program is named *yylex*. The *yylex* program will recognize expressions in a stream (called *input* in this memo) and perform the specified actions for each expression as it is detected. See Figure 1.

For a trivial example, consider a program to delete from the input all blanks or tabs at the ends of lines.

```
%%
[ \t]+$    ;
```

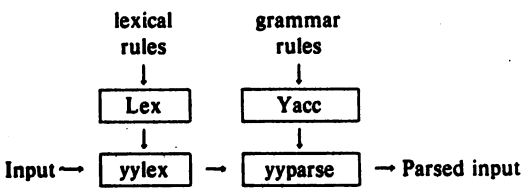
is all that is required. The program contains a %% delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression which matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C language convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates "one or more ..."; and the $ indicates "end of line," as in QED. No action is specified, so the program generated by Lex (yylex) will ignore these characters. Everything else will be copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

```
%%
[ \t]+$    ;
[ \t]+     printf(" ");
```

The finite automaton generated for this source will scan for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a newline character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule all remaining strings of blanks or tabs.

Lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. Lex can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface Lex and Yacc [3]. Lex programs recognize only regular expressions; Yacc writes parsers that accept a large class of context free grammars, but require a lower level analyzer to recognize input tokens. Thus, a combination of Lex and Yacc is often appropriate. When used as a preprocessor for a later parser generator, Lex is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. The flow of control in such a case (which might be the first half of a compiler, for example) is shown in Figure 2. Additional programs, written by other generators or by hand, can be added easily to programs written by Lex. Yacc users will realize that the name *yylex* is what Yacc expects its lexical analyzer to be named, so that the use of this name by Lex simplifies interfacing.

Lex generates a deterministic finite automaton from the regular expressions in the source [4]. The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time



Lex with Yacc

Figure 2

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 (in C) or branches of a computed GOTO (in Ratfor). The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.

Lex is not limited to source which can be interpreted on the basis of one character lookahead. For example, if there are two rules, one looking for ***ab*** and another for ***abcdefg***, and the input stream is ***abcdefh***, Lex will recognize ***ab*** and leave the input pointer just before ***cd***. . . Such backup is more costly than the processing of simpler languages.

2 Lex Source.

The general format of Lex source is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where the definitions and the user subroutines are often omitted. The second **%%** is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus

```
%%
```

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

In the outline of Lex programs shown above, the ***rules*** represent the user's control decisions; they are a table, in which the left column contains ***regular expressions*** (see section 3) and the right column contains ***actions,*** program fragments to be executed when the expressions are recognized. Thus an individual rule might appear

```
integer    printf("found keyword INT");
```

to look for the string ***integer*** in the input stream and print the message "found keyword INT" whenever it appears. In this example the host procedural language is C and the C library function ***printf*** is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces. As a slightly more useful example, suppose it is desired to change a number of words from British to American spelling. Lex rules such as

```
colour      printf("color");
mechanise   printf("mechanize");
petrol      printf("gas");
```

would be a start. These rules are not quite enough, since the word ***petroleum*** would become ***gaseum***; a way of dealing with this will be described later.

3 Lex Regular Expressions.

The definitions of regular expressions are very similar to those in QED [5]. A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; thus the regular expression

```
integer
```

matches the string ***integer*** wherever it appears and the expression

```
a57D
```

looks for the string ***a57D***.

Operators. The operator characters are

```
" \ [ ] ^ - ? . * + | ( ) $ / { } % < >
```

and if they are to be used as text characters, an escape should be used. The quotation mark operator (") indicates that whatever is contained between a pair of quotes is to be taken as text characters. Thus

```
xyz"++"
```

matches the string *xyz++* when it appears. Note that a part of a string may be quoted. It is harmless but unnecessary to quote an ordinary text character; the expression

```
"xyz++"
```

is the same as the one above. Thus by quoting every non-alphanumeric character being used as a text character, the user can avoid remembering the list above of current operator characters, and is safe should further extensions to Lex lengthen the list.

An operator character may also be turned into a text character by preceding it with \ as in

```
xyz\+\+
```

which is another, less readable, equivalent of the above

expressions. Another use of the quoting mechanism is to get a blank into an expression; normally, as explained above, blanks or tabs end a rule. Any blank character not contained within [] (see below) must be quoted. Several normal C escapes with \ are recognized: \n is newline, \t is tab, and \b is backspace. To enter \ itself, use \\. Since newline is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.

Character classes. Classes of characters can be specified using the operator pair []. The construction ***[ab]*** matches a single character, which may be ***a***, ***b***, or ***c***. Within square brackets, most operator meanings are ignored. Only three characters are special: these are \ – and ^. The – character indicates ranges. For example,

[a–z0–9<>_]

indicates the character class containing all the lower case letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using – between any pair of characters which are not both upper case letters, both lower case letters, or both digits is implementation dependent and will get a warning message. (E.g., [0-z] in ASCII is many more characters than it is in EBCDIC). If it is desired to include the character – in a character class, it should be first or last; thus

[–+0–9]

matches all the digits and the two signs.

In character classes, the ^ operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the computer character set. Thus

[^abc]

matches all characters except a, b, or c, including all special or control characters; or

[^a-zA-Z]

is any character which is not a letter. The \ character provides the usual escapes within character class brackets.

Arbitrary character. To match almost any character, the operator character

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

[\40-\176]

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

Optional expressions. The operator ? indicates an optional element of an expression. Thus

ab?c

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

Repeated expressions. Repetitions of classes are indicated by the operators * and +.

*a**

is any number of consecutive ***a*** characters, including zero; while

a+

is one or more instances of ***a***. For example,

[a-z]+

is all strings of lower case letters. And

[A–Za–z][A–Za–z0–9]*

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

Alternation and Grouping. The operator | indicates alternation:

(ab|cd)

matches either ***ab*** or ***cd***. Note that parentheses are used for grouping, although they are not necessary on the outside level;

ab|cd

would have sufficed. Parentheses can be used for more complex expressions:

(ab|cd+)?(ef)*

matches such strings as ***abefef***, ***efefef***, ***cdef***, or ***cddd***; but not ***abc***, ***abcd***, or ***abcdef***.

Context sensitivity. Lex will recognize a small amount of surrounding context. The two simplest operators for this are ^ and $. If the first character of an expression is ^, the expression will only be matched at the beginning of a line (after a newline character, or at the beginning of the input stream). This can never conflict with the other meaning of ^, complementation of character classes, since that only applies within the [] operators. If the very last character is $, the expression will only be matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the / operator character, which indicates trailing context. The expression

ab/cd

matches the string ***ab***, but only if followed by ***cd***. Thus

```
ab$
```

is the same as

```
ab/\n
```

Left context is handled in Lex by *start conditions* as explained in section 10. If a rule is only to be executed when the Lex automaton interpreter is in start condition *x*, the rule should be prefixed by

```
<x>
```

using the angle bracket operator characters. If we considered "being at the beginning of a line" to be start condition *ONE*, then the ^ operator would be equivalent to

```
<ONE>
```

Start conditions are explained more fully later.

Repetitions and Definitions. The operators {} specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example

```
{digit}
```

looks for a predefined string named *digit* and inserts it at that point in the expression. The definitions are given in the first part of the Lex input, before the rules. In contrast,

```
a{1,5}
```

looks for 1 to 5 occurrences of *a*.

Finally, initial % is special, being the separator for Lex source segments.

4 Lex Actions.

When an expression written as above is matched, Lex executes the corresponding action. This section describes some features of Lex which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus the Lex user who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When Lex is being used with Yacc, this is the normal situation. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted. Also, a character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

One of the simplest things that can be done is to ignore the input. Specifying a C null statement, ; as an action causes this result. A frequent rule is

```
[ \t\n]
```

which causes the three spacing characters (blank, tab, and newline) to be ignored.

Another easy way to avoid writing actions is the action character |, which indicates that the action for this rule is the action for the next rule. The previous example could also have been written

```
" "
"\t"
"\n"
```

with the same result, although in different style. The quotes around \n and \t are not required.

In more complex actions, the user will often want to know the actual text that matched some expression like *[a–z]+*. Lex leaves this text in an external character array named *yytext*. Thus, to print the name found, a rule like

```
[a-z]+    printf("%s", yytext);
```

will print the string in *yytext*. The C function *printf* accepts a format argument and data to be printed; in this case, the format is "print string" (% indicating data conversion, and *s* indicating string type), and the data are the characters in *yytext*. So this just places the matched string on the output. This action is so common that it may be written as ECHO:

```
[a-z]+    ECHO;
```

is the same as the above. Since the default action is just to print the characters found, one might ask why give a rule, like this one, which merely specifies the default action? Such rules are often required to avoid matching some other rule which is not desired. For example, if there is a rule which matches *read* it will normally match the instances of *read* contained in *bread* or *readjust*; to avoid this, a rule of the form *[a–z]+* is needed. This is explained further below.

Sometimes it is more convenient to know the end of what has been found; hence Lex also provides a count *yyleng* of the number of characters matched. To count both the number of words and the number of characters in words in the input, the user might write

```
[a-zA-Z]+    {words++; chars += yyleng;}
```

which accumulates in *chars* the number of characters in the words recognized. The last character in the string matched can be accessed by

```
yytext[yyleng-1]
```

in C or

```
yytext(yyleng)
```

in Ratfor.

Occasionally, a Lex action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, ***yymore()*** can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string would overwrite the current entry in ***yytext***. Second, ***yyless (n)*** may be called to indicate that not all the characters matched by the currently successful expression are wanted right now. The argument ***n*** indicates the number of characters in ***yytext*** to be retained. Further characters previously matched are returned to the input. This provides the same sort of lookahead offered by the / operator, but in a different form.

Example: Consider a language which defines a string as a set of characters between quotation (") marks, and provides that to include a " in a string it must be preceded by a \. The regular expression which matches that is somewhat confusing, so that it might be preferable to write

```
\"[^"]*    {
             if (yytext[yyleng-1] == '\\')
                yymore();
             else
                ... normal user processing
           }
```

which will, when faced with a string such as "***abc\"def***" first match the five characters "***abc***; then the call to ***yymore()*** will cause the next part of the string, "***def***, to be tacked on the end. Note that the final quote terminating the string should be picked up in the code labeled "normal processing".

The function ***yyless()*** might be used to reprocess text in various circumstances. Consider the C problem of distinguishing the ambiguity of "=-a". Suppose it is desired to treat this as "=- a" but print a message. A rule might be

```
=-[a-zA-Z]   {
             printf("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. There is another important routine in Ratfor, named ***lexshf***, which is described below under "Character Set". These routines define the relationship between external files and internal characters, and must all be retained or modified consistently. They may be redefined, to cause input or output to be transmitted to or from strange places, including other programs or internal memory; but the character set used must be consistent in all routines; a value of zero returned by ***input*** must mean end of file; and the relationship between ***unput*** and ***input*** must be retained or the Lex lookahead will not work. Lex does not look ahead at all if it does not have to, but every rule ending in + * ? or $ or containing / implies lookahead. Lookahead is also necessary to match an expression that is a prefix of another expression. See below for a discussion of the character set used by Lex. The standard Lex library imposes a 100 character limit on backup.

Another Lex library routine that the user will sometimes want to redefine is ***yywrap()*** which is called whenever Lex reaches an end-of-file. If ***yywrap*** returns a 1, Lex continues with the normal wrapup on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new source. In this case, the user should provide a ***yywrap*** which arranges for new input and returns 0. This instructs Lex to continue processing. The default ***yywrap*** always returns 1.

This routine is also a convenient place to print tables, summaries, etc. at the end of a program. Note that it is not possible to write a normal rule which recognizes end-of-file; the only access to this condition is through ***yywrap***. In fact, unless a private version of ***input()*** is supplied a file containing nulls cannot be handled, since a value of 0 returned by ***input*** is taken to be end-of-file.

In Ratfor all of the standard I/O library routines, ***input,***

output, unput, yywrap, and ***lexshf,*** are defined as integer functions. This requires ***input*** and ***yywrap*** to be called with arguments. One dummy argument is supplied and ignored.

5 Ambiguous Source Rules.

Lex can handle ambiguous specifications. When more than one expression can match the current input, Lex chooses as follows:

1) The longest match is preferred.
2) Among rules which matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules

```
integer    keyword action ...;
[a-z]+     identifier action ...;
```

to be given in that order. If the input is ***integers***, it is taken as an identifier, because ***[a-z]+*** matches 8 characters while ***integer*** matches only 7. If the input is ***integer***, both rules match 7 characters, and the keyword rule is selected because it was given first. Anything shorter (e.g. ***int***) will not match the expression ***integer*** and so the identifier interpretation is used.

The principle of preferring the longest match makes rules containing expressions like .* dangerous. For example,

```
'.*'
```

might seem a good way of recognizing a string in single quotes. But it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input

```
'first' quoted string here, 'second' here
```

the above expression will match

```
'first' quoted string here, 'second'
```

which is probably not what was wanted. A better rule is of the form

```
'[^\n]*'
```

which, on the above input, will stop after ***'first'***. The consequences of errors like this are mitigated by the fact that the . operator will not match newline. Thus expressions like .* stop on the current line. Don't try to defeat this with expressions like ***[.\n]+*** or equivalents; the Lex generated program will try to read the entire input file, causing internal buffer overflows.

Note that Lex is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for once and only once. For example, suppose it is desired to count occurrences of both ***she*** and ***he*** in an input text. Some Lex rules to do this might be

```
she    s++;
he     h++;
\n     |
.      ;
```

where the last two rules ignore everything besides ***he*** and ***she***. Remember that . does not include newline. Since ***she*** includes ***he***, Lex will normally ***not*** recognize the instances of ***he*** included in ***she***, since once it has passed a ***she*** those characters are gone.

Sometimes the user would like to override this choice. The action REJECT means "go do the next alternative." It causes whatever rule was second choice after the current rule to be executed. The position of the input pointer is adjusted accordingly. Suppose the user really wants to count the included instances of ***he***:

```
she    {s++; REJECT;}
he     {h++; REJECT;}
\n     |
.      ;
```

these rules are one way of changing the previous example to do just that. After counting each expression, it is rejected; whenever appropriate, the other expression will then be counted. In this example, of course, the user could note that ***she*** includes ***he*** but not vice versa, and omit the REJECT action on ***he***; in other cases, however, it would not be possible a priori to tell which input characters were in both classes.

Consider the two rules

```
a[bc]+     { ... ; REJECT;}
a[cd]+     { ... ; REJECT;}
```

If the input is ***ab***, only the first rule matches, and on ***ad*** only the second matches. The input string ***accb*** matches the first rule for four characters and then the second rule for three characters. In contrast, the input ***accd*** agrees with the second rule for four characters and then the first rule for three.

In general, REJECT is useful whenever the purpose of Lex is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other. Suppose a digram table of the input is desired; normally the digrams overlap, that is the word ***the*** is considered to contain both ***th*** and ***he***. Assuming a two-dimensional array named ***digram*** to be incremented, the appropriate source is

```
%%
[a-z][a-z]    {digram[yytext[0]][yytext[1]]++; REJECT;}
\n            ;
```

where the REJECT is necessary to pick up a letter pair beginning at every character, rather than at every other character.

6 Lex Source Definitions.

Remember the format of the Lex source:

```
{definitions}
%%
{rules}
%%
{user routines}
```

So far only the rules have been described. The user needs additional options, though, to define variables for use in his program and for use by Lex. These can go either in the definitions section or in the rules section.

Remember that Lex is turning the rules into a program. Any source not intercepted by Lex is copied into the generated program. There are three classes of such things.

1) Any line which is not part of a Lex rule or action which begins with a blank or tab is copied into the Lex generated program. Such source input prior to the first %% delimiter will be external to any function in the code; if it appears immediately after the first %%, it appears in an appropriate place for declarations in the function written by Lex which contains the actions. This material must look like program fragments, and should precede the first Lex rule.

   As a side effect of the above, lines which begin with a blank or tab, and which contain a comment, are passed through to the generated program. This can be used to include comments in either the Lex source or the generated code. The comments should follow the host language convention.

2) Anything included between lines containing only %{ and %} is copied out as above. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column 1, or copying lines that do not look like programs.

3) Anything after the third %% delimiter, regardless of formats, etc., is copied out after the Lex output.

Definitions intended for Lex are given before the first %% delimiter. Any line in this section not contained between %{ and %}, and begining in column 1, is assumed to define Lex substitution strings. The format of such lines is

```
name translation
```

and it causes the string given as a translation to be associated with the name. The name and translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be called out by the {name} syntax in a rule. Using {D} for the digits and {E} for an exponent field, for example, might abbreviate rules to recognize numbers:

```
D                       [0-9]
E                       [TEde][-+]?{D}+
%%
{D}+                    printf("integer");
{D}+"."{D}*({E})?       |
{D}*"."{D}+({E})?       |
{D}+{E}
```

Note the first two rules for real numbers; both require a decimal point and contain an optional exponent field, but the first requires at least one digit before the decimal point and the second requires at least one digit after the decimal point. To correctly handle the problem posed by a Fortran expression such as *35.EQ.I*, which does not contain a real number, a context-sensitive rule such as

```
[0-9]+/"."EQ    printf("integer");
```

could be used in addition to the normal rule for integers.

The definitions section may also contain other commands, including the selection of a host language, a character set table, a list of start conditions, or adjustments to the default size of arrays within Lex itself for larger source programs. These possibilities are discussed below under "Summary of Source Format," section 12.

7 Usage.

There are two steps in compiling a Lex source program. First, the Lex source must be turned into a generated program in the host general purpose language. Then this program must be compiled and loaded, usually with a library of Lex subroutines. The generated program is on a file named *lex.yy.c* for a C host language source and *lex.yy.r* for a Ratfor host environment. There are two I/O libraries, one for C defined in terms of the C standard library [6], and the other defined in terms of Ratfor. To indicate that a Lex source file is intended to be used with the Ratfor host language, make the first line of the file *%R*.

The C programs generated by Lex are slightly different on OS/370, because the OS compiler is less powerful than the UNIX or GCOS compilers, and does less at compile time. C programs generated on GCOS and UNIX are the same. The C host language is default, but may be explicitly requested by making the first line of the source file *%C*.

The Ratfor generated by Lex is the same on all systems, but can not be compiled directly on TSO. See below for instructions. The Ratfor I/O library, however, varies slightly because the different Fortrans disagree on the method of indicating end-of-input and the name of the library routine for logical AND. The Ratfor I/O library, dependent on Fortran character I/O, is quite slow. In particular it reads all input lines as 80A1 format; this will truncate any longer line, discarding your data, and pads any shorter line with blanks. The library version of *input* removes the padding (including any trailing blanks from the original input) before processing. Each source

file using a Ratfor host should begin with the "%R" command.

UNIX. The libraries are accessed by the loader flags *-llc* for C and *-llr* for Ratfor; the C name may be abbreviated to *-ll*. So an appropriate set of commands is

C Host	Ratfor Host
lex source	lex source
cc lex.yy.c -ll -lS	rc -2 lex.yy.r -llr

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. Note the "-2" option in the Ratfor compile command; this requests the larger version of the compiler, a useful precaution.

GCOS. The Lex commands on GCOS are stored in the "." library. The appropriate command sequences are:

C Host	Ratfor Host
./lex source	./lex source
./cc lex.yy.c ./lexclib h=	./rc a= lex.yy.r ./lexrlib h=

The resulting program is placed on the usual file *.program* for later execution (as indicated by the "h=" option); it may be copied to a permanent file if desired. Note the "a=" option in the Ratfor compile command; this indicates that the Fortran compiler is to run in ASCII mode.

TSO. Lex is just barely available on TSO. Restrictions imposed by the compilers which must be used with its output make it rather inconvenient. To use the C version, type

```
exec 'dot.lex.clist(lex)' 'sourcename'
exec 'dot.lex.clist(cload)' 'libraryname membername'
```

The first command analyzes the source file and writes a C program on file *lex.yy.text.* The second command runs this file through the C compiler and links it with the Lex C library (stored on 'hr289.lcl.load') placing the object program in your file *libraryname.LOAD(membername)* as a completely linked load module. The compiling command uses a special version of the C compiler command on TSO which provides an unusually large intermediate assembler file to compensate for the unusual bulk of C-compiled Lex programs on the OS system. Even so, almost any Lex source program is too big to compile, and must be split.

The same Lex command will compile Ratfor Lex programs, leaving a file *lex.yy.rat* instead of *lex.yy.text* in your directory. The Ratfor program must be edited, however, to compensate for peculiarities of IBM Ratfor. A command sequence to do this, and then compile and load, is available. The full commands are:

```
exec 'dot.lex.clist(lex)' 'sourcename'
exec 'dot.lex.clist(rload)' 'libraryname membername'
```

with the same overall effect as the C language commands. However, the Ratfor commands will run in a 150K byte partition, while the C commands require 250K bytes to operate.

The steps involved in processing the generated Ratfor program are:

a. Edit the Ratfor program.
1. Remove all tabs.
2. Change all lower case letters to upper case letters.
3. Convert the file to an 80-column card image file.

b. Process the Ratfor through the Ratfor preprocessor to get Fortran code.

c. Compile the Fortran.

d. Load with the libraries 'hr289.lrl.load' and 'sys1.fortlib'.

The final load module will only read input in 80-character fixed length records. ***Warning:*** Work is in progress on the IBM C compiler, and Lex and its availability on the IBM 370 are subject to change without notice.

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 -lS
```

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]+      {
            scanf(-1, yytext, "%d", &k);
            if (k%7 == 0)
                printf("%d", k+3);
            else
                printf("%d",k);
            }
```

to do just that. The rule [0-9]+ recognizes strings of digits; *scanf* 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]+                {
                        scanf(-1, yytext, "%d", &k);
                        printf("%d", k%7 == 0 ? k+3 : k);
                        }
-?[0-9.]+               ECHO;
[A-Za-z][A-Za-z0-9]+    ECHO;
```

Numerical strings containing a "." or preceded by a letter will be picked up by one of the last two rules, and not changed. The *if-else* has been replaced by a C conditional expression to save space; the form *a?b:c* means "if *a* then *b* else *c*".

For an example of statistics gathering, here is a program which histograms the lengths of words, where a word is defined as a string of letters.

```
            int lengs[100];
%%
[a-z]+      lengs[yyleng]++;
.           |
\n          ;
%%
yywrap()
{
int i;
printf("Length  No. words\n");
for(i=0; i<100; i++)
    if (lengs[i] > 0)
        printf("%5d%10d\n",i,lengs[i]);
return(1);
}
```

This program accumulates the histogram, while producing no output. At the end of the input it prints the table. The final statement *return(1);* indicates that Lex is to perform wrapup. If *yywrap* returns zero (false) it implies that further input is available and the program is to continue reading and processing. To provide a *yywrap* that never returns true causes an infinite loop.

As a larger example, here are some parts of a program written by N. L. Schryer to convert double precision Fortran to single precision Fortran. Because Fortran does not distinguish upper and lower case letters, this routine begins by defining a set of classes including both cases of each letter:

```
a       [aA]
b       [bB]
c       [cC]
...
z       [zZ]
```

An additional class recognizes white space:

```
W       [ \t]*
```

The first rule changes "double precision" to "real", or "DOUBLE PRECISION" to "REAL".

```
{d}{o}{u}{b}{l}{e}{W}{p}{r}{e}{c}{i}{s}{i}{o}{n} {
    printf(yytext[0]=='d'? "real" : "REAL");
    }
```

Care is taken throughout this program to preserve the case (upper or lower) of the original program. The conditional operator is used to select the proper form of the keyword. The next rule copies continuation card indications to avoid confusing them with constants:

```
^"      "[^ 0]      ECHO;
```

In the regular expression, the quotes surround the blanks. It is interpreted as "beginning of line, then five blanks, then anything but blank or zero." Note the two different meanings of ^. There follow some rules to change double precision constants to ordinary floating constants.

```
[0-9]+{W}{d}{W}[+-]?{W}[0-9]+       |
[0-9]+{W}"."{W}{d}{W}[+-]?{W}[0-9]+ |
"."{W}[0-9]+{W}{d}{W}[+-]?{W}[0-9]+ {
    /* convert constants */
    for(p=yytext; *p != 0; p++)
        {
        if (*p == 'd' | *p == 'D')
            *p=+ 'e'- 'd';
        ECHO;
        }
```

After the floating point constant is recognized, it is scanned by the *for* loop to find the letter *d* or *D*. The program than adds *'e'-'d'*, which converts it to the next letter of the alphabet. The modified constant, now single-precision, is written out again. There follow a series of names which must be respelled to remove their initial *d*. By using the array *yytext* the same action suffices for all the names (only a sample of a rather long list is given here).

```
{d}{s}{i}{n}        |
{d}{c}{o}{s}        |
{d}{s}{q}{r}{t}     |
{d}{a}{t}{a}{n}     |
...
{d}{f}{l}{o}{a}{t}  printf("%s",yytext+1);
```

Another list of names must have initial *d* changed to initial *a*:

```
{d}{l}{o}{g}      |
{d}{l}{o}{g}10    |
{d}{m}{i}{n}1     |
{d}{m}{a}{x}1     {
                  yytext[0] =+ 'a' - 'd';
                  ECHO;
                  }
```

And one routine must have initial *d* changed to initial *r*:

```
{d}1{m}{a}{c}{h}    {yytext[0] =+ 'r' - 'd';
```

To avoid such names as ***dsinx*** being detected as instances of ***dsin***, some final rules pick up longer words as identifiers and copy some surviving characters:

```
[A-Za-z][A-Za-z0-9]*  |
[0-9]+                |
\n                    |
.                     ECHO;
```

Note that this program is not complete; it does not deal with the spacing problems in Fortran or with the use of keywords as identifiers.

10 Left Context Sensitivity.

Sometimes it is desirable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The ^ operator, for example, is a prior context operator, recognizing immediately preceding left context just as $ recognizes immediately following right context. Adjacent left context could be extended, to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often the relevant left context appeared some time earlier, such as at the beginning of a line.

This section describes three means of dealing with different environments: a simple use of flags, when only a few rules change from one environment to another, the use of ***start conditions*** on rules, and the possibility of making multiple lexical analyzers all run together. In each case, there are rules which recognize the need to change the environment in which the following input text is analyzed, and set some parameter to reflect the change. This may be a flag explicitly tested by the user's action code; such a flag is the simplest way of dealing with the problem, since Lex is not involved at all. It may be more convenient, however, to have Lex remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It will only be recognized when Lex is in that start condition. The current start condition may be changed at any time. Finally, if the sets of rules for the different environments are very dissimilar, clarity may be best achieved by writing several distinct lexical analyzers, and switching from one to another as desired.

Consider the following problem: copy the input to the output, changing the word ***magic*** to ***first*** on every line which began with the letter ***a***, changing ***magic*** to ***second*** on every line which began with the letter ***b***, and changing ***magic*** to ***third*** on every line which began with the letter ***c***. All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a flag:

```
            int flag;
%%
^a          {flag = 'a'; ECHO;}
^b          {flag = 'b'; ECHO;}
^c          {flag = 'c'; ECHO;}
\n          {flag =  0 ; ECHO;}
magic       {
            switch (flag)
            {
            case 'a': printf("first"); break;
            case 'b': printf("second"); break;
            case 'c': printf("third"); break;
            default: ECHO; break;
            }
            }
```

should be adequate.

To handle the same problem with start conditions, each start condition must be introduced to Lex in the definitions section with a line reading

```
%Start     name1 name2 ...
```

where the conditions may be named in any order. The word ***Start*** may be abbreviated to *s* or *S*. The conditions may be referenced at the head of a rule with the <> brackets:

```
<name1>expression
```

is a rule which is only recognized when Lex is in the start condition ***name1***. To enter a start condition, execute the action statement

```
BEGIN name1;
```

which changes the start condition to ***name1***. To resume the normal state,

```
BEGIN 0;
```

resets the initial condition of the Lex automaton interpreter. A rule may be active in several start conditions:

```
<name1,name2,name3>
```

is a legal prefix. Any rule not beginning with the <> prefix operator is always active.

The same example as before can be written:

```
%START AA BB CC
%%
^a                  {ECHO; BEGIN AA;}
^b                  {ECHO; BEGIN BB;}
^c                  {ECHO; BEGIN CC;}
\n                  {ECHO; BEGIN 0;}
<AA>magic           printf("first");
<BB>magic           printf("second");
<CC>magic           printf("third");
```

where the logic is exactly the same as in the previous method of handling the problem, but Lex does the work rather than the user's code.

11 Character Set.

The programs generated by Lex handle character I/O only through the routines ***input, output,*** and ***unput.*** Thus the character representation provided in these routines is accepted by Lex and employed to return values in ***yytext.*** For internal use a character is represented as a small integer which, if the standard library is used, has a value equal to the integer value of the bit pattern representing the character on the host computer. In C, the I/O routines are assumed to deal directly in this representation. In Ratfor, it is anticipated that many users will prefer left-adjusted rather than right-adjusted characters; thus the routine ***lexshf*** is called to change the representation delivered by ***input*** into a right-adjusted integer. If the user changes the I/O library, the routine ***lexshf*** should also be changed to a compatible version. The Ratfor library I/O system is arranged to represent the letter ***a*** as in the Fortran value ***1Ha*** while in C the letter ***a*** is represented as the character constant ***'a'***. If this interpretation is changed, by providing I/O routines which translate the characters, Lex must be told about it, by giving a translation table. This table must be in the definitions section, and must be bracketed by lines containing only "%T". The table contains lines of the form

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

which indicate the value associated with each character. Thus the next example maps the lower and upper case letters together into the integers 1 through 26, newline into 27, + and - into 28 and 29, and the digits into 30 through 39. Note the escape for newline. If a table is supplied, every character that is to appear either in the

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

Sample character table.

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.

It is not likely that C users will wish to use the character table feature; but for Fortran portability it may be essential.

Although the contents of the Lex Ratfor library routines for input and output run almost unmodified on UNIX, GCOS, and OS/370, they are not really machine independent, and would not work with CDC or Burroughs Fortran compilers. The user is of course welcome to replace ***input, output, unput*** and ***lexshf*** but to replace them by completely portable Fortran routines is likely to cause a substantial decrease in the speed of Lex Ratfor programs. A simple way to produce portable routines would be to leave ***input*** and ***output*** as routines that read with 80A1 format, but replace ***lexshf*** by a table lookup routine.

12 Summary of Source Format.

The general form of a Lex source file is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

The definitions section contains a combination of

1) Definitions, in the form "name space translation".
2) Included code, in the form "space code".
3) Included code, in the form

```
%{
code
%}
```

4) Start conditions, given in the form

%S name1 name2 ...

5) Character set tables, in the form

%T
number space character-string
...
%T

6) A language specifier, which must also precede any rules or included code, in the form "%C" for C or "%R" for Ratfor.

7) Changes to internal array sizes, in the form

%x nnn

where ***nnn*** is a decimal integer representing an array size and ***x*** selects the parameter as follows:

Letter	Parameter
p	positions
n	states
e	tree nodes
a	transitions
k	packed character classes
o	output array size

Lines in the rules section have the form "expression action" where the action may be continued on succeeding lines by using braces to delimit it.

Regular expressions in Lex use the following operators:

x	the character "x"
"x"	an "x", even if x is an operator.
\x	an "x", even if x is an operator.
[xy]	the character x or y.
[x-z]	the characters x, y or z.
[^x]	any character but x.
.	any character but newline.
^x	an x at the beginning of a line.
<y>x	an x when Lex is in start condition y.
x$	an x at the end of a line.
x?	an optional x.
x*	0,1,2, ... instances of x.
x+	1,2,3, ... instances of x.
x|y	an x or a y.
(x)	an x.
x/y	an x but only if followed by y.
{xx}	the translation of xx from the definitions section.
x{m,n}	***m*** through ***n*** occurrences of x

13 Caveats and Bugs.

There are pathological expressions which produce exponential growth of the tables when converted to deterministic machines; fortunately, they are rare.

REJECT does not rescan the input; instead it remembers the results of the previous scan. This means that if a rule with trailing context is found, and REJECT executed, the user must not have used ***unput*** to change the characters forthcoming from the input stream. This is the only restriction on the user's ability to manipulate the not-yet-processed input.

TSO Lex is an older version. Among the non-supported features are REJECT, start conditions, or variable length trailing context, And any significant Lex source is too big for the IBM C compiler when translated.

14 Acknowledgments.

As should be obvious from the above, the outside of Lex is patterned on Yacc and the inside on Aho's string matching routines. Therefore, both S. C. Johnson and A. V. Aho are really originators of much of Lex, as well as debuggers of it. Many thanks are due to both.

The code of the current version of Lex was designed, written, and debugged by Eric Schmidt.

15 References.

1. B. W. Kernighan and D. M. Ritchie, ***The C Programming Language,*** Prentice-Hall, N. J. (1978).
2. B. W. Kernighan, ***Ratfor: A Preprocessor for a Rational Fortran,*** Software – Practice and Experience, **5**, pp. 395-496 (1975).
3. S. C. Johnson, ***Yacc: Yet Another Compiler Compiler,*** Computing Science Technical Report No. 32, 1975, Bell Laboratories, Murray Hill, NJ 07974.
4. A. V. Aho and M. J. Corasick, ***Efficient String Matching: An Aid to Bibliographic Search,*** Comm. ACM **18,** 333-340 (1975).
5. B. W. Kernighan, D. M. Ritchie and K. L. Thompson, ***QED Text Editor,*** Computing Science Technical Report No. 5, 1972, Bell Laboratories, Murray Hill, NJ 07974.
6. D. M. Ritchie, private communication. See also M. E. Lesk, ***The Portable C Library,*** Computing Science Technical Report No. 31, Bell Laboratories, Murray Hill, NJ 07974.

The M4 Macro Processor

Brian W. Kernighan

Dennis M. Ritchie

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.

July 1, 1977

†UNIX is a Trademark of Bell Laboratories.

The M4 Macro Processor

Brian W. Kernighan

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

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

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

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

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

M4 provides a collection of about twenty built-in macros which perform various useful operations; in addition, the user can define new macros. Built-ins and user-defined macros work exactly the same way, except that some of the built-in macros have side effects on the state of the process.

Usage

On UNIX, use

m4 [files]

Each argument file is processed in order; if there are no arguments, or if an argument is '−', the standard input is read at that point. The processed text is written on the standard output, which may be captured for subsequent processing with

m4 [files] >outputfile

On GCOS, usage is identical, but the program is called **./m4**.

Defining Macros

The primary built-in function of M4 is **define**, which is used to define new macros. The input

define(name, stuff)

causes the string **name** to be defined as **stuff**. All subsequent occurrences of **name** will be replaced by **stuff**. **name** must be alphanumeric and must begin with a letter (the underscore _ counts as a letter). **stuff** is any text that contains balanced parentheses; it may stretch over multiple lines.

Thus, as a typical example,

```
define(N, 100)
...
if (i > N)
```

defines N to be 100, and uses this "symbolic

constant" in a later **if** statement.

The left parenthesis must immediately follow the word **define**, to signal that **define** has arguments. If a macro or built-in name is not followed immediately by '(', it is assumed to have no arguments. This is the situation for N above; it is actually a macro with no arguments, and thus when it is used there need be no (...) following it.

You should also notice that a macro name is only recognized as such if it appears surrounded by non-alphanumerics. For example, in

```
define(N, 100)
...
if (NNN > 100)
```

the variable NNN is absolutely unrelated to the defined macro N, even though it contains a lot of N's.

Things may be defined in terms of other things. For example,

```
define(N, 100)
define(M, N)
```

defines both M and N to be 100.

What happens if N is redefined? Or, to say it another way, is M defined as N or as 100? In M4, the latter is true — M is 100, so even if N subsequently changes, M does not.

This behavior arises because M4 expands macro names into their defining text as soon as it possibly can. Here, that means that when the string N is seen as the arguments of **define** are being collected, it is immediately replaced by 100; it's just as if you had said

```
define(M, 100)
```

in the first place.

If this isn't what you really want, there are two ways out of it. The first, which is specific to this situation, is to interchange the order of the definitions:

```
define(M, N)
define(N, 100)
```

Now M is defined to be the string N, so when you ask for M later, you'll always get the value of N at that time (because the M will be replaced by N which will be replaced by 100).

Quoting

The more general solution is to delay the expansion of the arguments of **define** by *quoting* them. Any text surrounded by the single quotes ` and ' is not expanded immediately, but has the quotes stripped off. If you say

```
define(N, 100)
define(M, `N')
```

the quotes around the N are stripped off as the argument is being collected, but they have served their purpose, and M is defined as the string N, not 100. The general rule is that M4 always strips off one level of single quotes whenever it evaluates something. This is true even outside of macros. If you want the word **define** to appear in the output, you have to quote it in the input, as in

```
`define' = 1;
```

As another instance of the same thing, which is a bit more surprising, consider redefining N:

```
define(N, 100)
...
define(N, 200)
```

Perhaps regrettably, the N in the second definition is evaluated as soon as it's seen; that is, it is replaced by 100, so it's as if you had written

```
define(100, 200)
```

This statement is ignored by M4, since you can only define things that look like names, but it obviously doesn't have the effect you wanted. To really redefine N, you must delay the evaluation by quoting:

```
define(N, 100)
...
define(`N', 200)
```

In M4, it is often wise to quote the first argument of a macro.

If ` and ' are not convenient for some reason, the quote characters can be changed with the built-in **changequote**:

```
changequote([, ])
```

makes the new quote characters the left and right brackets. You can restore the original characters with just

changequote

There are two additional built-ins related to **define**. **undefine** removes the definition of some macro or built-in:

```
undefine(`N')
```

removes the definition of N. (Why are the quotes absolutely necessary?) Built-ins can be removed with **undefine**, as in

```
undefine(`define')
```

but once you remove one, you can never get it back.

The built-in **ifdef** provides a way to determine if a macro is currently defined. In particular, M4 has pre-defined the names **unix** and **gcos** on the corresponding systems, so you can tell which one you're using:

```
ifdef(`unix', `define(wordsize,16)' )
ifdef(`gcos', `define(wordsize,36)' )
```

makes a definition appropriate for the particular machine. Don't forget the quotes!

ifdef actually permits three arguments; if the name is undefined, the value of **ifdef** is then the third argument, as in

```
ifdef(`unix', on UNIX, not on UNIX)
```

Arguments

So far we have discussed the simplest form of macro processing — replacing one string by another (fixed) string. User-defined macros may also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its **define**) any occurrence of **$n** will be replaced by the **n**th argument when the macro is actually used. Thus, the macro **bump**, defined as

```
define(bump, $1 = $1 + 1)
```

generates code to increment its argument by 1:

```
bump(x)
```

is

```
x = x + 1
```

A macro can have as many arguments as you want, but only the first nine are accessible, through **$1** to **$9**. (The macro name itself is **$0**, although that is less commonly used.) Arguments that are not supplied are replaced by null strings, so we can define a macro **cat** which simply concatenates its arguments, like this:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

Thus

```
cat(x, y, z)
```

is equivalent to

```
xyz
```

$4 through **$9** are null, since no corresponding arguments were provided.

Leading unquoted blanks, tabs, or newlines that occur during argument collection are discarded. All other white space is retained. Thus

```
define(a,  b  c)
```

defines **a** to be **b c**.

Arguments are separated by commas, but parentheses are counted properly, so a comma "protected" by parentheses does not terminate an argument. That is, in

```
define(a, (b,c))
```

there are only two arguments; the second is literally **(b,c)**. And of course a bare comma or parenthesis can be inserted by quoting it.

Arithmetic Built-ins

M4 provides two built-in functions for doing arithmetic on integers (only). The simplest is **incr**, which increments its numeric argument by 1. Thus to handle the common programming situation where you want a variable to be defined as "one more than N", write

```
define(N, 100)
define(N1, `incr(N)')
```

Then **N1** is defined as one more than the current value of **N**.

The more general mechanism for arithmetic is a built-in called **eval**, which is capable of arbitrary arithmetic on integers. It provides the operators (in decreasing order of precedence)

```
unary + and -
** or ^        (exponentiation)
* / % (modulus)
+ -
== != < <= > >=
!             (not)
& or &&       (logical and)
| or ||       (logical or)
```

Parentheses may be used to group operations where needed. All the operands of an expression given to **eval** must ultimately be numeric. The numeric value of a true relation (like 1>0) is 1, and false is 0. The precision in **eval** is 32 bits on UNIX and 36 bits on GCOS.

As a simple example, suppose we want M to be 2**N+1. Then

```
define(N, 3)
define(M, `eval(2**N+1)')
```

As a matter of principle, it is advisable to quote the defining text for a macro unless it is very simple indeed (say just a number); it usually gives the result you want, and is a good habit to get into.

File Manipulation

You can include a new file in the input at any time by the built-in function **include**:

```
include(filename)
```

inserts the contents of **filename** in place of the **include** command. The contents of the file is often a set of definitions. The value of **include** (that is, its replacement text) is the contents of the file; this can be captured in definitions, etc.

It is a fatal error if the file named in **include** cannot be accessed. To get some control over this situation, the alternate form **sinclude** can be used; **sinclude** ("silent include") says nothing and continues if it can't access the file.

It is also possible to divert the output of M4 to temporary files during processing, and output the collected material upon command. M4 maintains nine of these diversions, numbered 1 through 9. If you say

```
divert(n)
```

all subsequent output is put onto the end of a temporary file referred to as **n**. Diverting to this file is stopped by another **divert** command; in particular, **divert** or **divert(0)** resumes the normal output process.

Diverted text is normally output all at once at the end of processing, with the diversions output in numeric order. It is possible, however, to bring back diversions at any time, that is, to append them to the current diversion.

```
undivert
```

brings back all diversions in numeric order, and **undivert** with arguments brings back the selected diversions in the order given. The act of undiverting discards the diverted stuff, as does diverting into a diversion whose number is not between 0 and 9 inclusive.

The value of **undivert** is *not* the diverted stuff. Furthermore, the diverted material is *not* rescanned for macros.

The built-in **divnum** returns the number of the currently active diversion. This is zero during normal processing.

System Command

You can run any program in the local operating system with the **syscmd** built-in. For example,

```
syscmd(date)
```

on UNIX runs the **date** command. Normally **syscmd** would be used to create a file for a subsequent **include**.

To facilitate making unique file names, the built-in **maketemp** is provided, with specifications identical to the system function *mktemp:* a string of XXXXX in the argument is replaced by the process id of the current process.

Conditionals

There is a built-in called **ifelse** which enables you to perform arbitrary conditional testing. In the simplest form,

```
ifelse(a, b, c, d)
```

compares the two strings **a** and **b**. If these are identical, **ifelse** returns the string **c**; otherwise it returns **d**. Thus we might define a macro called **compare** which compares two strings and returns "yes" or "no" if they are the same or different.

define(compare, `ifelse($1, $2, yes, no)')

Note the quotes, which prevent too-early evaluation of **ifelse**.

If the fourth argument is missing, it is treated as empty.

ifelse can actually have any number of arguments, and thus provides a limited form of multi-way decision capability. In the input

ifelse(a, b, c, d, e, f, g)

if the string **a** matches the string **b**, the result is **c**. Otherwise, if **d** is the same as **e**, the result is **f**. Otherwise the result is **g**. If the final argument is omitted, the result is null, so

ifelse(a, b, c)

is **c** if **a** matches **b**, and null otherwise.

String Manipulation

The built-in **len** returns the length of the string that makes up its argument. Thus

len(abcdef)

is 6, and **len((a,b))** is 5.

The built-in **substr** can be used to produce substrings of strings. **substr(s, i, n)** returns the substring of **s** that starts at the **i**th position (origin zero), and is **n** characters long. If **n** is omitted, the rest of the string is returned, so

substr(`now is the time', 1)

is

ow is the time

If **i** or **n** are out of range, various sensible things happen.

index(s1, s2) returns the index (position) in **s1** where the string **s2** occurs, or −1 if it doesn't occur. As with **substr**, the origin for strings is 0.

The built-in **translit** performs character transliteration.

translit(s, f, t)

modifies **s** by replacing any character found in **f** by the corresponding character of **t**. That is,

translit(s, aeiou, 12345)

replaces the vowels by the corresponding digits. If **t** is shorter than **f**, characters which don't have an entry in **t** are deleted; as a limiting case, if **t** is not present at all, characters from **f** are deleted from **s**. So

translit(s, aeiou)

deletes vowels from **s**.

There is also a built-in called **dnl** which deletes all characters that follow it up to and including the next newline; it is useful mainly for throwing away empty lines that otherwise tend to clutter up M4 output. For example, if you say

```
define(N, 100)
define(M, 200)
define(L, 300)
```

the newline at the end of each line is not part of the definition, so it is copied into the output, where it may not be wanted. If you add **dnl** to each of these lines, the newlines will disappear.

Another way to achieve this, due to J. E. Weythman, is

```
divert(-1)
    define(...)
    ...
divert
```

Printing

The built-in **errprint** writes its arguments out on the standard error file. Thus you can say

errprint(`fatal error')

dumpdef is a debugging aid which dumps the current definitions of defined terms. If there are no arguments, you get everything; otherwise you get the ones you name as arguments. Don't forget to quote the names!

Summary of Built-ins

Each entry is preceded by the page number where it is described.

3 changequote(L, R)
1 define(name, replacement)
4 divert(number)
4 divnum
5 dnl
5 dumpdef(`name', `name', ...)
5 errprint(s, s, ...)
4 eval(numeric expression)
3 ifdef(`name', this if true, this if false)
5 ifelse(a, b, c, d)
4 include(file)
3 incr(number)
5 index(s1, s2)
5 len(string)
4 maketemp(...XXXXX...)
4 sinclude(file)
5 substr(string, position, number)
4 syscmd(s)
5 translit(str, from, to)
3 undefine(`name')
4 undivert(number,number,...)

Acknowledgements

We are indebted to Rick Becker, John Chambers, Doug McIlroy, and especially Jim Weythman, whose pioneering use of M4 has led to several valuable improvements. We are also deeply grateful to Weythman for several substantial contributions to the code.

References

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

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.** UNIX communicates with IBM's Job Entry Subsystem by mimicking an IBM 360 remote multileaving work station. The *UNIX User's Manual* page *rje*(8) summarizes its design and operation. The manual also contains a description of the *send*(1) command, which is the user's primary method of submitting jobs to RJE, and *rjestat*(1), 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*. This 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.

** In this paper, RJE refers to the UNIX facilities and *not* to the Remote Job Entry feature of IBM's HASP or 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*(1) 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*(1) 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 page 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*(1).

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 pathname, unless it refers to an executable file (see D above). For cases A, B, and C above (and case D, if a full pathname is not supplied), the name of the user's login directory will be used to form a full pathname.

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*(1) 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*(1) 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*(1) manual page 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 Administrative Guide

1. INTRODUCTION

This document reflects the Plexus implementation of RJE, and is based on the *Remote Job Entry Administrative Guide* from Bell Laboratories, by M. J. Fitton.

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 Plexus Sys3 UNIX system, one Intelligent Communications Processor (ICP) is needed per remote line to drive the RJE line.

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)

---

* UNIX is a Trademark of AT&T Bell Laboratories.

1. In this document, RJE refers to the UNIX facilities of *rje*(8) and *not* to the Remote Job Entry feature of IBM's HASP or JES2 subsystems.

- 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*(1) 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 associated with 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, but the delimiters are present. 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*(1) 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**.

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 rje1disp
-rwxr-xr-x   2 rje      rje         2328 Mar  4 10:21 rje1halt
-rwxr-xr-x   2 rje      rje        10396 Apr 15 10:07 rje1init
-r-x------   2 rje      rje          785 Apr  8 09:00 rje1load
-rwsr-xr-x   2 rje      rje         5040 Mar 27 09:28 rje1qer
-rwxr-xr-x   2 rje      rje         4072 Apr  1 15:40 rje1recv
-rwxr-xr-x   2 rje      rje         3888 Mar 27 09:35 rje1xmit
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 *rje1*∗, *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 *rje1* to log errors. It can be useful for debugging *rje1* problems.
- **joblog** – used by *rje1qer* and *rjestat* to notify *rje1xmit* that a job (or console request) has been submitted. It also contains the process-group number of the *rje1* 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 *rje1halt* has been executed. The existence of this file indicates to *rjestat* that *rje1* has been halted by the operator.
- **testjob** – a sample job that can be submitted to test the *rje1* subsystem. Originally, the job control statements may have to be changed to suit your IBM system.

When *rje1* terminates abnormally, the file **dead** should appear in this directory. This file contains a short message indicating why *rje1* 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 *rje1disp*.
- **spool** – used by *send* to store temporary files to be submitted to the remote machine. This directory must be mode 777.
- **squeue** – used by *rje1* to store submitted files until they are transmitted. The program *rje1qer* 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 ic0"
su rje –c "/usr/rje2/rje2load ic1"
```

The file **/usr/rje/sque/log** is removed to ensure the correct operation of *shqer*. When invoked, *rjeload* performs the following steps:

1. Finds the proper ICP device by using the minor device number of the corresponding VPM device (the first two bits).
2. Uses *dnld*(1) to perform the following:
   - reset the ICP
   - load the VPM script (**/etc/rjeproto**)
   - start the ICP 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. Truncates the console response file **resp**.
2. 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.
3. Sets up pipes for process communication.
4. Resets process-group for RJE subsystem and restarts error logging.
5. Rebuilds the **joblog** file from jobs queued for transmission.
6. Notifies *rjedisp* (via a pipe) of any returned files still remaining in the **rpool** directory.
7. Starts the appropriate background processes (*rje?xmit*, *rje?recv*, and *rje?disp*).
8. 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. 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.

2. **Reader** refers to the logical readers used by RJE.
3. **Console** refers to the RJE logical console, which is distinct from the logical readers.

- **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 and 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 files to **size** blocks if a received file exceeds this value (**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 occurences. 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 ICP 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 ICP 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 **–s***host* 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*(1).

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.

Since **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
- ICP/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. Section 2.3 describes this file in detail.
3. ICP/VPM setup – the VPM software must be installed and the proper VPM and ICP devices made. Each VPM device must correspond to the proper ICP 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 ICP has been set up, the remote system, or the hardware.

7.4 ICP/VPM Problems

This section describes the ICP and VPM uses, and problems that can occur. After installing ICP hardware and making ICP devices, all VPM software and devices must be made. See *vpm*(4). The following is a snapshot of the ICP and VPM devices used on our hypothetical machine:

```
crw-r--r--   1 rje      rje          9,   0 Apr 16 07:04 /dev/ic0
crw-r--r--   1 rje      rje         15,   0 Apr 16 10:51 /dev/vpm0

crw-r--r--   1 rje      rje          9,   1 Apr 10 08:21 /dev/ic1
crw-r--r--   1 rje      rje         15,  41 Apr  7 13:25 /dev/vpm1
```

where **/dev/ic**? corresponds to **/dev/vpm**? (?=0,1). The VPM minor device number determines which VPM and ICP devices are used. See *vpm*(4) to determine VPM minor device numbers. The program *rjeload* prints the devices being 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 ICP address and interrupt vector are correct.
2. Proper Devices – the major and minor device numbers for both the ICP and VPM must be correct. It should also be verified that the RJE subsystem is using the correct ICP 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 ICP and VPM script (see Section 5.1). If no output appears from *snoop* when *rjeload* is executed, either the ICP is not working properly, or the ICP 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    * ICP 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    * ICP 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     * ICP 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 ICP 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
     .                   *
     .                   * More of the same
     .                   *
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
     .                   *
     .                   * And so on
     .                   *
```

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 since 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     *    .
76   TR      TIMEOUT   *    .
77   TR      S-NAK     *    .
78   TR      TIMEOUT   *    .
79   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   * ICP 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 type of thing 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.

SED — A Non-interactive Text Editor

Lee E. McMahon

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Sed is a non-interactive context editor that runs on the UNIX† operating system. *Sed* is designed to be especially useful in three cases:

1) To edit files too large for comfortable interactive editing;
2) To edit any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode.
3) To perform multiple 'global' editing functions efficiently in one pass through the input.

This memorandum constitutes a manual for users of *sed.*

August 15, 1978

†UNIX is a Trademark of Bell Laboratories.

SED — A Non-interactive Text Editor

Lee E. McMahon

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

Sed is a non-interactive context editor designed to be especially useful in three cases:

1) To edit files too large for comfortable interactive editing;
2) To edit any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode;
3) To perform multiple 'global' editing functions efficiently in one pass through the input.

Since only a few lines of the input reside in core at one time, and no temporary files are used, the effective size of file that can be edited is limited only by the requirement that the input and output fit simultaneously into available secondary storage.

Complicated editing scripts can be created separately and given to *sed* as a command file. For complex edits, this saves considerable typing, and its attendant errors. *Sed* running from a command file is much more efficient than any interactive editor known to the author, even if that editor can be driven by a pre-written script.

The principal loss of functions compared to an interactive editor are lack of relative addressing (because of the line-at-a-time operation), and lack of immediate verification that a command has done what was intended.

Sed is a lineal descendant of the UNIX editor, *ed.* Because of the differences between interactive and non-interactive operation, considerable changes have been made between *ed* and *sed;* even confirmed users of *ed* will frequently be surprised (and probably chagrined), if they rashly use *sed* without reading Sections 2 and 3 of this document. The most striking family resemblance between the two editors is in the class of patterns ('regular expressions') they recognize; the code for matching patterns is copied almost verbatim from the code for *ed,* and the description of regular expressions in Section 2 is copied almost verbatim from the UNIX Programmer's Manual[1]. (Both code and description were written by Dennis M. Ritchie.)

1. Overall Operation

Sed by default copies the standard input to the standard output, perhaps performing one or more editing commands on each line before writing it to the output. This behavior may be modified by flags on the command line; see Section 1.1 below.

The general format of an editing command is:

[address1,address2][function][arguments]

One or both addresses may be omitted; the format of addresses is given in Section 2. Any number of blanks or tabs may separate the addresses from the function. The function must be present; the available commands are discussed in Section 3. The arguments may be required or optional, according to which function is given; again, they are discussed in Section 3 under each individual function.

Tab characters and spaces at the beginning of lines are ignored.

1.1. Command-line Flags

Three flags are recognized on the command line:

-n: tells *sed* not to copy all lines, but only those specified by *p* functions or *p* flags after *s* functions (see Section 3.3);

-e: tells *sed* to take the next argument as an editing command;

-f: tells *sed* to take the next argument as a file name; the file should contain editing commands, one to a line.

1.2. Order of Application of Editing Commands

Before any editing is done (in fact, before any input file is even opened), all the editing commands are compiled into a form which will be moderately efficient during the execution phase (when the commands are actually applied to lines of the input file). The commands are compiled in the order in which they are encountered; this is generally the order in which they will be attempted at execution time. The commands are applied one at a time; the input to each command is the output of all preceding commands.

The default linear order of application of editing commands can be changed by the flow-of-control commands, *t* and *b* (see Section 3). Even when the order of application is changed by these commands, it is still true that the input line to any command is the output of any previously applied command.

1.3. Pattern-space

The range of pattern matches is called the pattern space. Ordinarily, the pattern space is one line of the input text, but more than one line can be read into the pattern space by using the *N* command (Section 3.6.).

1.4. Examples

Examples are scattered throughout the text. Except where otherwise noted, the examples all assume the following input text:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

(In no case is the output of the *sed* commands to be considered an improvement on Coleridge.)

Example:

The command

```
2q
```

will quit after copying the first two lines of the input. The output will be:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
```

2. ADDRESSES: Selecting lines for editing

Lines in the input file(s) to which editing commands are to be applied can be selected by addresses. Addresses may be either line numbers or context addresses.

The application of a group of commands can be controlled by one address (or address-pair) by grouping the commands with curly braces ('{ }')(Sec. 3.6.).

2.1. Line-number Addresses

A line number is a decimal integer. As each line is read from the input, a line-number counter is incremented; a line-number address matches (selects) the input line which causes the internal counter to equal the address line-number. The counter runs cumulatively through multiple input files; it is not reset when a new input file is opened.

As a special case, the character $ matches the last line of the last input file.

2.2. Context Addresses

A context address is a pattern ('regular expression') enclosed in slashes ('/'). The regular expressions recognized by *sed* are constructed as follows:

1) An ordinary character (not one of those discussed below) is a regular expression, and matches that character.
2) A circumflex '^' at the beginning of a regular expression matches the null character at the beginning of a line.
3) A dollar-sign '$' at the end of a regular expression matches the null character at the end of a line.
4) The characters '\n' match an imbedded newline character, but not the newline at the end of the pattern space.
5) A period '.' matches any character except the terminal newline of the pattern space.
6) A regular expression followed by an asterisk '*' matches any number (including 0) of adjacent occurrences of the regular expression it follows.
7) A string of characters in square brackets '[ ]' matches any character in the string, and no others. If, however, the first character of the string is circumflex '^', the regular expression matches any character *except* the characters in the string and the terminal newline of the pattern space.
8) A concatenation of regular expressions is a regular expression which matches the concatenation of strings matched by the components of the regular expression.
9) A regular expression between the sequences '\(' and '\)' is identical in effect to the unadorned regular expression, but has side-effects which are described under the *s* command below and specification 10) immediately below.
10) The expression '*d*' means the same string of characters matched by an expression enclosed in '\(' and '\)' earlier in the same pattern. Here *d* is a single digit; the string specified is that beginning with the *d*th occurrence of '\(' counting from the left. For example, the expression '^\(.*\)\1' matches a line beginning with two repeated occurrences of the same string.
11) The null regular expression standing alone (e.g., '//') is equivalent to the last regular expression compiled.

To use one of the special characters (^ $. * [ ] \ /) as a literal (to match an occurrence of itself in the input), precede the special character by a backslash '\'.

For a context address to 'match' the input requires that the whole pattern within the address match some portion of the pattern space.

2.3. Number of Addresses

The commands in the next section can have 0, 1, or 2 addresses. Under each command the maximum number of allowed addresses is given. For a command to have more addresses than the maximum allowed is considered an error.

If a command has no addresses, it is applied to every line in the input.

If a command has one address, it is applied to all lines which match that address.

If a command has two addresses, it is applied to the first line which matches the first address, and to all subsequent lines until (and including) the first subsequent line which matches the second address. Then an attempt is made on subsequent lines to again match the first address,

and the process is repeated.

Two addresses are separated by a comma.

Examples:

/an/	matches lines 1, 3, 4 in our sample text
/an.*an/	matches line 1
/^an/	matches no lines
/./	matches all lines
/\./	matches line 5
/r*an/	matches lines 1,3, 4 (number = zero!)
/\(an\).*\1/	matches line 1

3. FUNCTIONS

All functions are named by a single character. In the following summary, the maximum number of allowable addresses is given enclosed in parentheses, then the single character function name, possible arguments enclosed in angles (< >), an expanded English translation of the single-character name, and finally a description of what each function does. The angles around the arguments are *not* part of the argument, and should not be typed in actual editing commands.

3.1. Whole-line Oriented Functions

(2)d -- delete lines

The *d* function deletes from the file (does not write to the output) all those lines matched by its address(es).

It also has the side effect that no further commands are attempted on the corpse of a deleted line; as soon as the *d* function is executed, a new line is read from the input, and the list of editing commands is re-started from the beginning on the new line.

(2)n -- next line

The *n* function reads the next line from the input, replacing the current line. The current line is written to the output if it should be. The list of editing commands is continued following the *n* command.

(1)a\
<text> -- append lines

The *a* function causes the argument <text> to be written to the output after the line matched by its address. The *a* command is inherently multi-line; *a* must appear at the end of a line, and <text> may contain any number of lines. To preserve the one-command-to-a-line fiction, the interior newlines must be hidden by a backslash character ('\') immediately preceding the newline. The <text> argument is terminated by the first unhidden newline (the first one not immediately preceded by backslash).

Once an *a* function is successfully executed, <text> will be written to the output regardless of what later commands do to the line which triggered it. The triggering line may be deleted entirely; <text> will still be written to the output.

The <text> is not scanned for address matches, and no editing commands are attempted on it. It does not cause any change in the line-number counter.

(1)i\
<text> -- insert lines

The *i* function behaves identically to the *a* function, except that <text> is written to the output *before* the matched line. All other comments about the *a* function apply to the *i* function as well.

(2)c\
<text> -- change lines

The *c* function deletes the lines selected by its address(es), and replaces them with the lines in <text>. Like *a* and *i*, *c* must be followed by a newline hidden by a backslash; and interior new lines in <text> must be hidden by backslashes.

The *c* command may have two addresses, and therefore select a range of lines. If it does, all the lines in the range are deleted, but only one copy of <text> is written to the output, *not* one copy per line deleted. As with *a* and *i*, <text> is not scanned for address matches, and no editing commands are attempted on it. It does not change the line-number counter.

After a line has been deleted by a *c* function, no further commands are attempted on the corpse.

If text is appended after a line by *a* or *r* functions, and the line is subsequently changed, the text inserted by the *c* function will be placed *before* the text of the *a* or *r* functions. (The *r* function is described in Section 3.4.)

Note: Within the text put in the output by these functions, leading blanks and tabs will disappear, as always in *sed* commands. To get leading blanks and tabs into the output, precede the first desired blank or tab by a backslash; the backslash will not appear in the output.

Example:

The list of editing commands:

```
n
a\
XXXX
d
```

applied to our standard input, produces:

```
In Xanadu did Kubhla Khan
XXXX
Where Alph, the sacred river, ran
XXXX
Down to a sunless sea.
```

In this particular case, the same effect would be produced by either of the two following command lists:

```
n          n
i\         c\
XXXX       XXXX
d
```

3.2. Substitute Function

One very important function changes parts of lines selected by a context search within the line.

(2)s<pattern><replacement><flags> -- substitute

The *s* function replaces *part* of a line (selected by <pattern>) with <replacement>. It can best be read:

Substitute for <pattern>, <replacement>

The <pattern> argument contains a pattern, exactly like the patterns in addresses (see 2.2 above). The only difference between <pattern> and a context address is that the context address must be delimited by slash ('/') characters; <pattern> may be delimited by any character other than space or newline.

By default, only the first string matched by <pattern> is replaced, but see the *g* flag below.

The <replacement> argument begins immediately after the second delimiting character of <pattern>, and must be followed immediately by another instance of the delimiting character. (Thus there are exactly *three* instances of the delimiting character.)

The <replacement> is not a pattern, and the characters which are special in patterns do not have special meaning in <replacement>. Instead, other characters are special:

& is replaced by the string matched by <pattern>

d (where *d* is a single digit) is replaced by the *d*th substring matched by parts of <pattern> enclosed in '\(' and '\)'. If nested substrings occur in <pattern>, the *d*th is determined by counting opening delimiters ('\(').

As in patterns, special characters may be made literal by preceding them with backslash ('\').

The <flags> argument may contain the following flags:

g -- substitute <replacement> for all (non-overlapping) instances of <pattern> in the line. After a successful substitution, the scan for the next instance of <pattern> begins just after the end of the inserted characters; characters put into the line from <replacement> are not rescanned.

p -- print the line if a successful replacement was done. The *p* flag causes the line to be written to the output if and only if a substitution was actually made by the *s* function. Notice that if several *s* functions, each followed by a *p* flag, successfully substitute in the same input line, multiple copies of the line will be written to the output: one for each successful substitution.

w <filename> -- write the line to a file if a successful replacement was done. The *w* flag causes lines which are actually substituted by the *s* function to be written to a file named by <filename>. If <filename> exists before *sed* is run, it is overwritten; if not, it is created.

A single space must separate *w* and <filename>.

The possibilities of multiple, somewhat different copies of one input line being written are the same as for *p*.

A maximum of 10 different file names may be mentioned after *w* flags and *w* functions (see below), combined.

Examples:

The following command, applied to our standard input,

```
s/to/by/w changes
```

produces, on the standard output:

```
In Xanadu did Kubhla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless by man
Down by a sunless sea.
```

and, on the file 'changes':

```
Through caverns measureless by man
Down by a sunless sea.
```

If the nocopy option is in effect, the command:

```
s/[.,;?:]/*P&*/gp
```

produces:

```
A stately pleasure dome decree*P:*
Where Alph*P,* the sacred river*P,* ran
Down to a sunless sea*P.*
```

Finally, to illustrate the effect of the *g* flag, the command:

```
/X/s/an/AN/p
```

produces (assuming nocopy mode):

```
In XANadu did Kubhla Khán
```

and the command:

```
/X/s/an/AN/gp
```

produces:

```
In XANadu did Kubhla KhAN
```

3.3. Input-output Functions

(2)p -- print

The print function writes the addressed lines to the standard output file. They are written at the time the *p* function is encountered, regardless of what succeeding editing commands may do to the lines.

(2)w <filename> -- write on <filename>

The write function writes the addressed lines to the file named by <filename>. If the file previously existed, it is overwritten; if not, it is created. The lines are written exactly as they exist when the write function is encountered for each line, regardless of what subsequent editing commands may do to them.

Exactly one space must separate the *w* and <filename>.

A maximum of ten different files may be mentioned in write functions and *w* flags after *s* functions, combined.

(1)r <filename> -- read the contents of a file

The read function reads the contents of <filename>, and appends them after the line matched by the address. The file is read and appended regardless of what subsequent editing commands do to the line which matched its address. If *r* and *a* functions are executed on the same line, the text from the *a*

functions and the *r* functions is written to the output in the order that the functions are executed.

Exactly one space must separate the *r* and <filename>. If a file mentioned by a *r* function cannot be opened, it is considered a null file, not an error, and no diagnostic is given.

NOTE: Since there is a limit to the number of files that can be opened simultaneously, care should be taken that no more than ten files be mentioned in *w* functions or flags; that number is reduced by one if any *r* functions are present. (Only one read file is open at one time.)

Examples

Assume that the file 'note1' has the following contents:

Note: Kubla Khan (more properly Kublai Khan; 1216-1294) was the grandson and most eminent successor of Genghiz (Chingiz) Khan, and founder of the Mongol dynasty in China.

Then the following command:

/Kubla/r note1

produces:

In Xanadu did Kubla Khan
Note: Kubla Khan (more properly Kublai Khan; 1216-1294) was the grandson and most eminent successor of Genghiz (Chingiz) Khan, and founder of the Mongol dynasty in China.
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.

3.4. Multiple Input-line Functions

Three functions, all spelled with capital letters, deal specially with pattern spaces containing imbedded newlines; they are intended principally to provide pattern matches across lines in the input.

(2)N -- Next line

The next input line is appended to the current line in the pattern space; the two input lines are separated by an imbedded newline. Pattern matches may extend across the imbedded newline(s).

(2)D -- Delete first part of the pattern space

Delete up to and including the first newline character in the current pattern space. If the pattern space becomes empty (the only newline was the terminal newline), read another line from the input. In any case, begin the list of editing commands again from its beginning.

(2)P -- Print first part of the pattern space

Print up to and including the first newline in the pattern space.

The *P* and *D* functions are equivalent to their lower-case counterparts if there are no imbedded newlines in the pattern space.

3.5. Hold and Get Functions

Four functions save and retrieve part of the input for possible later use.

(2)h -- hold pattern space

The *h* functions copies the contents of the pattern space into a hold area (destroying the previous contents of the hold area).

(2)H -- Hold pattern space

The *H* function appends the contents of the pattern space to the contents of the hold area; the former and new contents are separated by a newline.

(2)g -- get contents of hold area

The *g* function copies the contents of the hold area into the pattern space (destroying the previous contents of the pattern space).

(2)G -- Get contents of hold area

The *G* function appends the contents of the hold area to the contents of the pattern space; the former and new contents are separated by a newline.

(2)x -- exchange

The exchange command interchanges the contents of the pattern space and the hold area.

Example

The commands

```
1h
1s/ did.*//
1x
G
s/\n/  :/
```

applied to our standard example, produce:

```
In Xanadu did Kubla Khan  :In Xanadu
A stately pleasure dome decree:  :In Xanadu
Where Alph, the sacred river, ran  :In Xanadu
Through caverns measureless to man  :In Xanadu
Down to a sunless sea.  :In Xanadu
```

3.6. Flow-of-Control Functions

These functions do no editing on the input lines, but control the application of functions to the lines selected by the address part.

(2)! -- Don't

The *Don't* command causes the next command (written on the same line), to be applied to all and only those input lines *not* selected by the adress part.

(2){ -- Grouping

The grouping command '{' causes the next set of commands to be applied (or not applied) as a block to the input lines selected by the addresses of the grouping command. The first of the commands under control of the grouping may appear on the same line as the '{' or on the next line.

The group of commands is terminated by a matching '}' standing on a line by itself.

Groups can be nested.

(0):<label> -- place a label

The label function marks a place in the list of editing commands which may be referred to by *b* and *t* functions. The <label> may be any sequence of eight or fewer characters; if two different colon functions have identical labels, a compile time diagnostic will be generated, and no execution attempted.

(2)b<label> -- branch to label

The branch function causes the sequence of editing commands being applied to the current input line to be restarted immediately after the place where a colon function with the same <label> was encountered. If no colon function with the same label can be found after all the editing commands have been compiled, a compile time diagnostic is produced, and no execution is attempted.

A *b* function with no <label> is taken to be a branch to the end of the list of editing commands; whatever should be done with the current input line is done, and another input line is read; the list of editing commands is restarted from the beginning on the new line.

(2)t<label> -- test substitutions

The *t* function tests whether *any* successful substitutions have been made on the current input line; if so, it branches to <label>; if not, it does nothing. The flag which indicates that a successful substitution has been executed is reset by:

1) reading a new input line, or
2) executing a *t* function.

3.7. Miscellaneous Functions

(1)= -- equals

The = function writes to the standard output the line number of the line matched by its address.

(1)q -- quit

The *q* function causes the current line to be written to the output (if it should be), any appended or read text to be written, and execution to be terminated.

Reference

[1] Ken Thompson and Dennis M. Ritchie, *The UNIX Programmer's Manual.* Bell Laboratories, 1978.

Source Code Control System User's Guide

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™ based PWB (Programmer's Workbench) time-sharing system.

This document, together with relevant portions of [1], 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.

Source Code Control System
User's Guide

LIST OF FIGURES

LIST OF TABLES

January 1980

1. INTRODUCTION

The Source Code Control System (SCCS) is a collection of PWB 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 in environments in which 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 [1], 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.
- *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 PWB system, create files, and use the text editor [1]. 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 [1]) 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".

---

1. All references of the form *name*(*N*) refer to item *name* in command writeuP section *N* of [1].

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:

```
1.2
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.

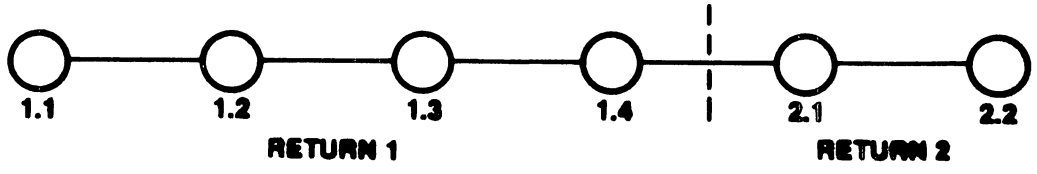


Figure 1. Evolution of an SCCS File

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.

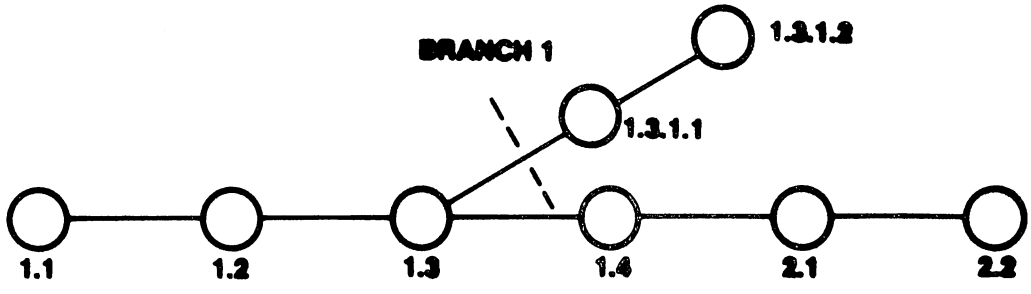


Figure 2. Tree Structure with Branch Deltas

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).

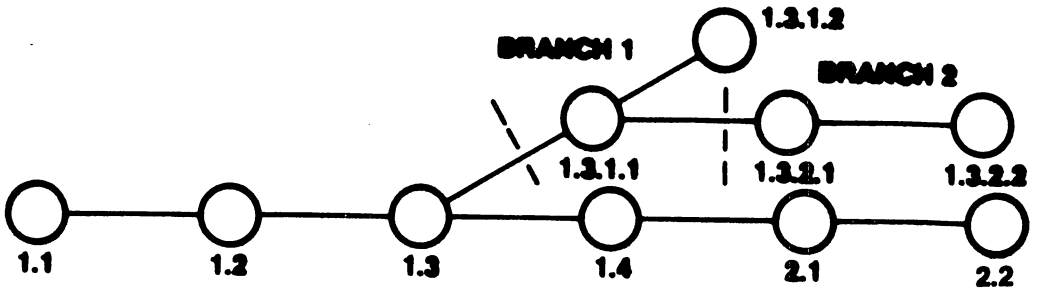


Figure 3. Extending the Branching Concept

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 [1] with, for example, the *find*(1) or *ls*(1) commands. Again, names of non-SCCS files and of unreadable files are silently ignored.

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 PWB 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 PWB system); this subject is further discussed in Section 6.1.

---

2. Because of Permission modes (see *chmod*(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* [1] 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 [1]) 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 *PWB 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 PWB 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 [1] 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]:

```
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 ≤ R ≤ 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* by inserting a line containing the above information. Before inserting this line, 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 [1].

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.*

---

3. Other information may be there also, 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.

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:

```
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-filename
```

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

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

if 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*(1) to make ID keyword substitutions before detecting and interpreting control lines, is necessary if *send*(1) 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.* Note that 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 newline character. The user's response may be up to 512 characters long, with newlines *not* intended to terminate the response escaped by "\".

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 are used to supply the commentary (comments and MR numbers, respectively) on the command line, rather than through the standard input. For example:

```
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.*

---

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

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.

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.

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

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.

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.

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".

---

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

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 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.

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

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:

```
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:

$$\text{floor} \leq R \leq \text{ceiling}$$

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* PWB 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 (\), newline, 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 which is a disjunction of the possible errors detected. That is, 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 PWB 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). The reason for this is that those commands that modify SCCS files do so by creating a temporary copy of the file (called 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, removing it and renaming the *x-file* would break the link. 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

10. Previously, the OPerating System under which SCCS executed allowed for only 256 unique user IDs. This Presented the situation in which several users needed to share user IDs (and thus shared identical file Permissions). The OPerating System currently in use (Version 7 of UNIX) allows for 65,536 unique user IDs, and it is recommended that each user have a unique user ID.

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's function is to invoke the desired SCCS command and to cause it to *inherit* the privileges of the interface program for the duration of that command's execution. In this manner, 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 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.* The project-dependent interface program, as its name implies, must be custom-built for each project.

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 PWB 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 PWB 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

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.

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 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 PWB operations group to request a restoral of the file 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.

REFERENCES

[1] Bell Laboratories, *Documents for Use with the PWB Time-Sharing System.*

ADDENDUM

The following changes to the Source Code Control System are effective with the UNIX SYSTEM III release.

1. **Modified Commands**

   Three SCCS commands have been modified:

   1. **comb**
   2. **get**
   3. **sccsdiff**

   Modifications to each of these commands are described below.

   1.1 **comb**

   - enhancement

     Comb generates a shell procedure that, when executed, reduces the size of an SCCS file. Because of temporary file naming conventions, two or more **comb**-generated shell procedures could not be executed concurrently. Temporary files are now uniquely named so that simultaneous execution is possible.

   1.2 **get**

   - enhancement

     Previously the *-i* and *-x* keyletters (for forced inclusion or exclusion of deltas to produce the generated file) would imply the *-k* keyletter. That is, the generated file would be created with mode 644 and identification keyword replacement would be suppressed. The *-i* and *-k* keyletters no longer imply the *-k* keyletter.

   - coding error correction

     Under certain circumstances, temporary files that should only have existed for the duration of the execution of **get** would not be removed when **get** terminated. Temporary files are now properly removed.

   1.3 **sccsdiff**

   - new capability

     A new keyletter (*-s*), which takes a numeric argument, allows the user to specify the file segmentation size that **bdiff**(1) (used by **sccsdiff**) will pass to **diff**(1). This can be useful when a high system load causes **diff** to fail due to lack of space.

   - change

     The output of **sccsdiff** is no longer piped through **pr**(1) by default. A new keyletter (*-n*) specifies that the output is to be piped through **pr** but arguments cannot be passed to **pr** as was the previous case. This alleviates **sccsdiff** knowing anything about **pr**.

2. **New Commands**

Two new commands have been added to SCCS:

sact print current SCCS file editing activity.

unget undo the effect opf a previous **get** for editing of an SCCS file.

The manual entries for these commands are provided in the *Plexus Sys3 UNIX Programmer's Manual -- Volume 1*.

Function and Use of an SCCS Interface Program

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*.

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)).

---

* 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.

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* pathname 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" (the ellipses represent other arguments that may 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 directory "/x1/xyz/sccs" as the one 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.

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);
}
```

April 1980

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 eighty UNIX† computer systems has been established using the telephone system as its primary communication medium. The network was designed to meet the growing demands for software distribution and exchange. Some advantages of our design are:

- The startup cost is low. A system needs only a dial-up port, but systems with automatic calling units have much more flexibility.
- No operating system changes are required to install or use the system.
- The communication is basically over dial-up lines, however, hardwired communication lines can be used to increase speed.
- The command for sending/receiving files is simple to use.

1. Purpose

The widespread use of the UNIX system[1] within Bell Laboratories has produced problems of software distribution and maintenance. A conventional mechanism was set up to distribute the operating system and associated programs from a central site to the various users. However this mechanism alone does not meet all software distribution needs. Remote sites generate much software and must transmit it to other sites. Some UNIX systems are themselves central sites for redistribution of a particular specialized utility, such as the Switching Control Center System. Other sites have particular, often long-distance needs for software exchange; switching research, for example, is carried on in New Jersey, Illinois, Ohio, and Colorado. In addition, general purpose utility programs are written at all UNIX system sites. The UNIX system is modified and enhanced by many people in many places and it would be very constricting to deliver new software in a one-way stream without any alternative for the user sites to respond with changes of their own.

Straightforward software distribution is only part of the problem. A large project may exceed the capacity of a single computer and several machines may be used by the one group of people. It then becomes necessary for them to pass messages, data and other information back an forth between computers.

Several groups with similar problems, both inside and outside of Bell Laboratories, have constructed networks built of hardwired connections only.[2,3] Our network, however, uses both dial-up and hardwired connections so that service can be provided to as many sites as possible.

---

†UNIX is a Trademark of Bell Laboratories.

2. Design Goals

Although some of our machines are connected directly, others can only communicate over low-speed dial-up lines. Since the dial-up lines are often unavailable and file transfers may take considerable time, we spool all work and transmit in the background. We also had to adapt to a community of systems which are independently operated and resistant to suggestions that they should all buy particular hardware or install particular operating system modifications. Therefore, we make minimal demands on the local sites in the network. Our implementation requires no operating system changes; in fact, the transfer programs look like any other user entering the system through the normal dial-up login ports, and obeying all local protection rules.

We distinguish "active" and "passive" systems on the network. Active systems have an automatic calling unit or a hardwired line to another system, and can initiate a connection. Passive systems do not have the hardware to initiate a connection. However, an active system can be assigned the job of calling passive systems and executing work found there; this makes a passive system the functional equivalent of an active system, except for an additional delay while it waits to be polled. Also, people frequently log into active systems and request copying from one passive system to another. This requires two telephone calls, but even so, it is faster than mailing tapes.

Where convenient, we use hardwired communication lines. These permit much faster transmission and multiplexing of the communications link. Dial-up connections are made at either 300 or 1200 baud; hardwired connections are asynchronous up to 9600 baud and might run even faster on special-purpose communications hardware.[4,5] Thus, systems typically join our network first as passive systems and when they find the service more important, they acquire automatic calling units and become active systems; eventually, they may install high-speed links to particular machines with which they handle a great deal of traffic. At no point, however, must users change their programs or procedures.

The basic operation of the network is very simple. Each participating system has a spool directory, in which work to be done (files to be moved, or commands to be executed remotely) is stored. A standard program, *uucico*, performs all transfers. This program starts by identifying a particular communication channel to a remote system with which it will hold a conversation. *Uucico* then selects a device and establishes the connection, logs onto the remote machine and starts the *uucico* program on the remote machine. Once two of these programs are connected, they first agree on a line protocol, and then start exchanging work. Each program in turn, beginning with the calling (active system) program, transmits everything it needs, and then asks the other what it wants done. Eventually neither has any more work, and both exit.

In this way, all services are available from all sites; passive sites, however, must wait until called. A variety of protocols may be used; this conforms to the real, non-standard world. As long as the caller and called programs have a protocol in common, they can communicate. Furthermore, each caller knows the hours when each destination system should be called. If a destination is unavailable, the data intended for it remain in the spool directory until the destination machine can be reached.

The implementation of this Bell Laboratories network between independent sites, all of which store proprietary programs and data, illustratives the pervasive need for security and administrative controls over file access. Each site, in configuring its programs and system files, limits and monitors transmission. In order to access a file a user needs access permission for the machine that contains the file and access permission for the file itself. This is achieved by first requiring the user to use his password to log into his local machine and then his local machine logs into the remote machine whose files are to be accessed. In addition, records are kept identifying all files that are moved into and out of the local system, and how the requester of such accesses identified himself. Some sites may arrange to permit users only to call up and request work to be done; the calling users are then called back before the work is actually done. It is then possible to verify that the request is legitimate from the standpoint of the target system, as well as the originating system. Furthermore, because of the call-back, no site can masquerade as another even if it knows all the necessary passwords.

Each machine can optionally maintain a sequence count for conversations with other machines and require a verification of the count at the start of each conversation. Thus, even if call back is not in use, a successful masquerade requires the calling party to present the correct sequence number. A would-be impersonator must not just steal the correct phone number, user name, and password, but also the sequence count, and must call in sufficiently promptly to precede the next legitimate request from either side. Even a successful masquerade will be detected on the next correct conversation.

3. Processing

The user has two commands which set up communications, *uucp* to set up file copying, and *uux* to set up command execution where some of the required resources (system and/or files) are not on the local machine. Each of these commands will put work and data files into the spool directory for execution by *uucp* daemons. Figure 1 shows the major blocks of the file transfer process.

File Copy

The *uucico* program is used to perform all communications between the two systems. It performs the following functions:

- Scan the spool directory for work.
- Place a call to a remote system.
- Negotiate a line protocol to be used.
- Start program *uucico* on the remote system.
- Execute all requests from both systems.
- Log work requests and work completions.

Uucico may be started in several ways;

a) by a system daemon,

b) by one of the *uucp* or *uux* programs,

c) by a remote system.

Scan For Work

The file names in the spool directory are constructed to allow the daemon programs *(uucico, uuxqt)* to determine the files they should look at, the remote machines they should call and the order in which the files for a particular remote machine should be processed.

Call Remote System

The call is made using information from several files which reside in the uucp program directory. At the start of the call process, a lock is set on the system being called so that another call will not be attempted at the same time.

The system name is found in a "systems" file. The information contained for each system is:

[1] system name,

[2] times to call the system (days-of-week and times-of-day),

[3] device or device type to be used for call,

[4] line speed,

[5] phone number,

[6] login information (multiple fields).

The time field is checked against the present time to see if the call should be made. The *phone number* may contain abbreviations (e.g. "nyc", "boston") which get translated into dial sequences using a "dial-codes" file. This permits the same "phone number" to be stored at every site, despite local variations in telephone services and dialing conventions.

A "devices" file is scanned using fields [3] and [4] from the "systems" file to find an available device for the connection. The program will try all devices which satisfy [3] and [4] until a connection is made, or no more devices can be tried. If a non-multiplexable device is successfully opened, a lock file is created so that another copy of *uucico* will not try to use it. If the connection is complete, the *login information* is used to log into the remote system. Then a command is sent to the remote system to start the *uucico* program. The conversation between the two *uucico* programs begins with a handshake started by the called, *SLAVE*, system. The *SLAVE* sends a message to let the *MASTER* know it is ready to receive the system identification and conversation sequence number. The response from the *MASTER* is verified by the *SLAVE* and if acceptable, protocol selection begins.

Line Protocol Selection

The remote system sends a message

Pproto-list

where *proto-list* is a string of characters, each representing a line protocol. The calling program checks the proto-list for a letter corresponding to an available line protocol and returns a *use-protocol* message. The *use-protocol* message is

Ucode

where code is either a one character protocol letter or a *N* which means there is no common protocol.

Greg Chesson designed and implemented the standard line protocol used by the uucp transmission program. Other protocols may be added by individual installations.

Work Processing

During processing, one program is the *MASTER* and the other is *SLAVE*. Initially, the calling program is the *MASTER*. These roles may switch one or more times during the conversation.

There are four messages used during the work processing, each specified by the first character of the message. They are

S send a file,
R receive a file,
C copy complete,
H hangup.

The *MASTER* will send *R* or *S* messages until all work from the spool directory is complete, at which point an *H* message will be sent. The *SLAVE* will reply with SY, SN, RY, RN, HY, HN, corresponding to *yes* or *no* for each request.

The send and receive replies are based on permission to access the requested file/directory. After each file is copied into the spool directory of the receiving system, a copy-complete message is sent by the receiver of the file. The message *CY* will be sent if the UNIX *cp* command, used to copy from the spool directory, is successful. Otherwise, a *CN* message is sent. The requests and results are logged on both systems, and, if requested, mail is sent to the user reporting completion (or the user can request status information from the log program at any time).

The hangup response is determined by the *SLAVE* program by a work scan of the spool directory. If work for the remote system exists in the *SLAVE's* spool directory, a *HN* message

is sent and the programs switch roles. If no work exists, an *HY* response is sent.

A sample conversation is shown in Figure 2.

Conversation Termination

When a *HY* message is received by the *MASTER* it is echoed back to the *SLAVE* and the protocols are turned off. Each program sends a final "OO" message to the other.

4. Present Uses

One application of this software is remote mail. Normally, a UNIX system user writes "mail dan" to send mail to user "dan". By writing "mail usg!dan" the mail is sent to user "dan" on system "usg".

The primary uses of our network to date have been in software maintenance. Relatively few of the bytes passed between systems are intended for people to read. Instead, new programs (or new versions of programs) are sent to users, and potential bugs are returned to authors. Aaron Cohen has implemented a "stockroom" which allows remote users to call in and request software. He keeps a "stock list" of available programs, and new bug fixes and utilities are added regularly. In this way, users can always obtain the latest version of anything without bothering the authors of the programs. Although the stock list is maintained on a particular system, the items in the stockroom may be warehoused in many places; typically each program is distributed from the home site of its author. Where necessary, uucp does remote-to-remote copies.

We also routinely retrieve test cases from other systems to determine whether errors on remote systems are caused by local misconfigurations or old versions of software, or whether they are bugs that must be fixed at the home site. This helps identify errors rapidly. For one set of test programs maintained by us, over 70% of the bugs reported from remote sites were due to old software, and were fixed merely by distributing the current version.

Another application of the network for software maintenance is to compare files on two different machines. A very useful utility on one machine has been Doug McIlroy's "diff" program which compares two text files and indicates the differences, line by line, between them.[6] Only lines which are not identical are printed. Similarly, the program "uudiff" compares files (or directories) on two machines. One of these directories may be on a passive system. The "uudiff" program is set up to work similarly to the inter-system mail, but it is slightly more complicated.

To avoid moving large numbers of usually identical files, *uudiff* computes file checksums on each side, and only moves files that are different for detailed comparison. For large files, this process can be iterated; checksums can be computed for each line, and only those lines that are different actually moved.

The "uux" command has been useful for providing remote output. There are some machines which do not have hard-copy devices, but which are connected over 9600 baud communication lines to machines with printers. The *uux* command allows the formatting of the printout on the local machine and printing on the remote machine using standard UNIX command programs.

5. Performance

Throughput, of course, is primarily dependent on transmission speed. The table below shows the real throughput of characters on communication links of different speeds. These numbers represent actual data transferred; they do not include bytes used by the line protocol for data validation such as checksums and messages. At the higher speeds, contention for the processors on both ends prevents the network from driving the line full speed. The range of speeds represents the difference between light and heavy loads on the two systems. If desired, operating system modifications can be installed that permit full use of even very fast links.

Nominal speed	Characters/sec.
300 baud	27
1200 baud	100-110
9600 baud	200-850

In addition to the transfer time, there is some overhead for making the connection and logging in ranging from 15 seconds to 1 minute. Even at 300 baud, however, a typical 5,000 byte source program can be transferred in four minutes instead of the 2 days that might be required to mail a tape.

Traffic between systems is variable. Between two closely related systems, we observed 20 files moved and 5 remote commands executed in a typical day. A more normal traffic out of a single system would be around a dozen files per day.

The total number of sites at present in the main network is 82, which includes most of the Bell Laboratories full-size machines which run the UNIX operating system. Geographically, the machines range from Andover, Massachusetts to Denver, Colorado.

Uucp has also been used to set up another network which connects a group of systems in operational sites with the home site. The two networks touch at one Bell Labs computer.

6. Further Goals

Eventually, we would like to develop a full system of remote software maintenance. Conventional maintenance (a support group which mails tapes) has many well-known disadvantages.[7] There are distribution errors and delays, resulting in old software running at remote sites and old bugs continually reappearing. These difficulties are aggravated when there are 100 different small systems, instead of a few large ones.

The availability of file transfer on a network of compatible operating systems makes it possible just to send programs directly to the end user who wants them. This avoids the bottleneck of negotiation and packaging in the central support group. The "stockroom" serves this function for new utilities and fixes to old utilities. However, it is still likely that distributions will not be sent and installed as often as needed. Users are justifiably suspicious of the "latest version" that has just arrived; all too often it features the "latest bug." What is needed is to address both problems simultaneously:

1. Send distributions whenever programs change.
2. Have sufficient quality control so that users will install them.

To do this, we recommend systematic regression testing both on the distributing and receiving systems. Acceptance testing on the receiving systems can be automated and permits the local system to ensure that its essential work can continue despite the constant installation of changes sent from elsewhere. The work of writing the test sequences should be recovered in lower counseling and distribution costs.

Some slow-speed network services are also being implemented. We now have inter-system "mail" and "diff," plus the many implied commands represented by "uux." However, we still need inter-system "write" (real-time inter-user communication) and "who" (list of people logged in on different systems). A slow-speed network of this sort may be very useful for speeding up counseling and education, even if not fast enough for the distributed data base applications that attract many users to networks. Effective use of remote execution over slow-speed lines, however, must await the general installation of multiplexable channels so that long file transfers do not lock out short inquiries.

7. Lessons

The following is a summary of the lessons we learned in building these programs.

1. By starting your network in a way that requires no hardware or major operating system changes, you can get going quickly.
2. Support will follow use. Since the network existed and was being used, system maintainers were easily persuaded to help keep it operating, including purchasing additional hardware to speed traffic.
3. Make the network commands look like local commands. Our users have a resistance to learning anything new: all the inter-system commands look very similar to standard UNIX system commands so that little training cost is involved.
4. An initial error was not coordinating enough with existing communications projects: thus, the first version of this network was restricted to dial-up, since it did not support the various hardware links between systems. This has been fixed in the current system.

Acknowledgements

We thank G. L. Chesson for his design and implementation of the packet driver and protocol, and A. S. Cohen, J. Lions, and P. F. Long for their suggestions and assistance.

References

1. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Bell Sys. Tech. J.* **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. J. W. Hunt and M. D. McIlroy, "An Algorithm for Differential File Comparison," Comp. Sci. Tech. Rep. No. 41, Bell Laboratories, Murray Hill, New Jersey (D).
7. F. P. Brooks, Jr., *The Mythical Man-Month,* Addison-Wesley, Reading, Mass. (1975).

January 1980

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 hardwired 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 hardwired 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 "-r1" 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 hardwired 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). Hardwired 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 dialcode 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 okay for the call. Note that a time range that spans 0000 is permitted, for example, 0800-0600 means all times are ok 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 hardwired device to be used for the call. For the hardwired case, the last part of the special file name is used (e.g., tty0).

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 hardwired devices, this field contains the same string as used for the *device* field.

login

The login information is given as a series of fields and subfields in the format

[expect send] ...

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, dialup 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 dialup 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 (reused) 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 1980

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.

July 31, 1978

Yacc: Yet Another Compiler-Compiler

Stephen C. Johnson

Bell Laboratories
Murray Hill, New Jersey 07974

0: Introduction

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a *parser*, calls the user-supplied low-level input routine (the *lexical analyzer*) to pick up the basic items (called *tokens*) from the input stream. These tokens are organized according to the input structure rules, called *grammar rules*; when one of these rules has been recognized, then user code supplied for this rule, an *action*, is invoked; actions have the ability to return values and make use of the values of other actions.

Yacc is written in a portable dialect of C[1] and the actions, and output subroutine, are in C as well. Moreover, many of the syntactic conventions of Yacc follow C.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

```
date : month_name day ',' year ;
```

Here, *date*, *month_name*, *day*, and *year* represent structures of interest in the input process; presumably, *month_name*, *day*, and *year* are defined elsewhere. The comma "," is enclosed in single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

```
July 4, 1776
```

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a *terminal symbol*, while the structure recognized by the parser is called a *nonterminal symbol*. To avoid confusion, terminal symbols will usually be referred to as *tokens*.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

```
month_name : 'J' 'a' 'n' ;
month_name : 'F' 'e' 'b' ;

    . . .

month_name : 'D' 'e' 'c' ;
```

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and *month_name* would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond Yacc's ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a

month_name was seen; in this case, *month_name* would be a token.

Literal characters such as "," must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is realively easy to add to the above example the rule

```
date : month '/' day '/' year ;
```

allowing

```
7 / 4 / 1776
```

as a synonym for

```
July 4, 1776
```

In most cases, this new rule could be "slipped in" to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, Yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self contradictory, or they may require a more powerful recognition mechanism than that available to Yacc. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While Yacc cannot handle all possible specifications, its power compares favorably with similar systems; moreover, the constructions which are difficult for Yacc to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid Yacc specifications for their input revealed errors of conception or design early in the program development.

The theory underlying Yacc has been described elsewhere.[2, 3, 4] Yacc has been extensively used in numerous practical applications, including *lint*,[5] the Portable C Compiler,[6] and a system for typesetting mathematics.[7]

The next several sections describe the basic process of preparing a Yacc specification; Section 1 describes the preparation of grammar rules, Section 2 the preparation of the user supplied actions associated with these rules, and Section 3 the preparation of lexical analyzers. Section 4 describes the operation of the parser. Section 5 discusses various reasons why Yacc may be unable to produce a parser from a specification, and what to do about it. Section 6 describes a simple mechanism for handling operator precedences in arithmetic expressions. Section 7 discusses error detection and recovery. Section 8 discusses the operating environment and special features of the parsers Yacc produces. Section 9 gives some suggestions which should improve the style and efficiency of the specifications. Section 10 discusses some advanced topics, and Section 11 gives acknowledgements. Appendix A has a brief example, and Appendix B gives a summary of the Yacc input syntax. Appendix C gives an example using some of the more advanced features of Yacc, and, finally, Appendix D describes mechanisms and syntax no longer actively supported, but provided for historical continuity with older versions of Yacc.

1: Basic Specifications

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. In addition, for reasons discussed in Section 3, it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections: the *declarations*, *(grammar)*

rules, and *programs*. The sections are separated by double percent "%%" marks. (The percent "%" is generally used in Yacc specifications as an escape character.)

In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal Yacc specification is

```
%%
rules
```

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* . . . */, as in C and PL/I.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A : BODY ;
```

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are Yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot ".", underscore "_", and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes "'". As in C, the backslash "\" is an escape character within literals, and all the C escapes are recognized. Thus

'\n'	newline
'\r'	return
'\''	single quote "'"
'\\'	backslash "\"
'\t'	tab
'\b'	backspace
'\f'	form feed
'\xxx'	"xxx" in octal

For a number of technical reasons, the NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, the vertical bar "|" can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

```
A     :     B C D ;
A     :     E F ;
A     :     G ;
```

can be given to Yacc as

```
A     :     B C D
      |     E F
      |     G
      ;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

```
empty : ;
```

Names representing tokens must be declared; this is most simply done by writing

```
%token  name1  name2 . . .
```

in the declarations section. (See Sections 3 , 5, and 6 for much more discussion). Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the *start symbol*, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the %start keyword:

```
%start  symbol
```

The end of the input to the parser is signaled by a special token, called the *endmarker*. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the endmarker is seen; it *accepts* the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate; see section 3, below. Usually the endmarker represents some reasonably obvious I/O status, such as "end-of-file" or "end-of-record".

2: Actions

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces "{" and "}". For example,

```
A       :       '(' B ')'
                        {       hello( 1, "abc" ); }
```

and

```
XXX     :       YYY ZZZ
                        {       printf("a message\n");
                                flag = 25; }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol "dollar sign" "$" is used as a signal to Yacc in this context.

To return a value, the action normally sets the pseudo-variable "$$" to some value. For example, an action that does nothing but return the value 1 is

```
{ $$ = 1; }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, . . ., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

```
A     :     B C D ;
```

for example, then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, consider the rule

```
expr  :     '(' expr ')' ;
```

The value returned by this rule is usually the value of the *expr* in parentheses. This can be indicated by

```
expr  :     '(' expr ')'        { $$ = $2 ; }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form

```
A     :     B ;
```

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. Yacc permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

```
A     :     B
                  { $$ = 1; }
            C
                  { x = $2;  y = $3; }
      ;
```

the effect is to set *x* to 1, and *y* to the value returned by C.

Actions that do not terminate a rule are actually handled by Yacc by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. Yacc actually treats the above example as if it had been written:

```
$ACT  :     /* empty */
                  { $$ = 1; }
      ;

A     :     B $ACT C
                  { x = $2;  y = $3; }
      ;
```

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function *node*, written so that the call

```
node( L, n1, n2 )
```

creates a node with label L, and descendants n1 and n2, and returns the index of the newly created node. Then parse tree can be built by supplying actions such as:

```
expr    :       expr '+' expr
                        { $$ = node( '+', $1, $3 ); }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks "%{" and "%}". These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{  int variable = 0;  %}
```

could be placed in the declarations section, making *variable* accessible to all of the actions. The Yacc parser uses only names beginning in "yy"; the user should avoid such names.

In these examples, all the values are integers: a discussion of values of other types will be found in Section 10.

3: Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called *yylex*. The function returns an integer, the *token number*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable *yylval*.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by Yacc, or chosen by the user. In either case, the "# define" mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the Yacc specification file. The relevant portion of the lexical analyzer might look like:

```
yylex(){
        extern int yylval;
        int c;
        ...
        c = getchar();
        ...
        switch( c ) {
                ...
        case '0':
        case '1':
          ...
        case '9':
                yylval = c-'0';
                return( DIGIT );
                ...
                }
        ...
```

The intent is to return a token number of DIGIT, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT will be defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of token names *if* or *while* will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name *error* is reserved for error

handling, and should not be used naively (see Section 7).

As mentioned above, the token numbers may be chosen by Yacc or by the user. In the default situation, the numbers are chosen by Yacc. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

To assign a token number to a token (including literals), the first appearance of the token name or literal *in the declarations section* can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user; thus, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the *Lex* program developed by Mike Lesk.[8] These lexical analyzers are designed to work in close harmony with Yacc parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. Lex can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) which do not fit any theoretical framework, and whose lexical analyzers must be crafted by hand.

4: How the Parser Works

Yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the *lookahead* token). The *current state* is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called *shift*, *reduce*, *accept*, and *error*. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls *yylex* to obtain the next token.
2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

The *shift* action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

```
IF      shift 34
```

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

The *reduce* action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a ".") is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

```
.       reduce 18
```

refers to *grammar rule* 18, while the action

```
IF      shift 34
```

refers to *state* 34.

Suppose the rule being reduced is

```
A       :       x y z   ;
```

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped equals the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack while recognizing *x*, *y*, and *z*, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left hand symbol and an ordinary shift of a token, however, so this action is called a *goto* action. In particular, the lookahead token is cleared by a shift, and is not affected by a goto. In any case, the uncovered state contains an entry such as:

```
A       goto 20
```

causing state 20 to be pushed onto the stack, and become the current state.

In effect, the reduce action "turns back the clock" in the parse, popping the states off the stack to go back to the state where the right hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right hand side of the rule is empty, no states are popped off of the stack: the uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable *yylval* is copied onto the value stack. After the return from the user code, the reduction is carried out. When the *goto* action is done, the external variable *yyval* is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The *accept* action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The *error* action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The parser reports an error, and attempts to recover the situation and resume parsing: the error recovery (as opposed to the detection of error) will be covered in Section 7.

It is time for an example! Consider the specification

```
%token DING DONG DELL
%%
rhyme :     sound place
      ;
sound :     DING DONG
      ;
place :     DELL
      ;
```

When Yacc is invoked with the **-v** option, a file called *y.output* is produced, with a human-readable description of the parser. The *y.output* file corresponding to the above grammar (with some statistics stripped off the end) is:

```
state 0
        $accept : _rhyme $end

        DING shift 3
        . error

        rhyme goto 1
        sound goto 2

state 1
        $accept : rhyme_$end

        $end accept
        . error

state 2
        rhyme : sound_place

        DELL shift 5
        . error

        place goto 4

state 3
        sound : DING_DONG

        DONG shift 6
        . error

state 4
        rhyme : sound place_   (1)

        . reduce 1

state 5
        place : DELL_   (3)

        . reduce 3

state 6
        sound : DING DONG_   (2)

        . reduce 2
```

Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The _ character is used to indicate what has been seen, and what is yet to come, in each rule. Suppose the input is

DING DONG DELL

It is instructive to follow the steps of the parser while processing this input.

Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, *DING*, is read, becoming the lookahead token. The action in state 0 on *DING* is is "shift 3", so state 3 is pushed onto the stack, and the lookahead token is cleared. State 3 becomes the current state. The next token, *DONG*, is read, becoming the lookahead token. The action in state 3 on the token *DONG* is

"shift 6", so state 6 is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the lookahead, the parser reduces by rule 2.

```
sound : DING DONG
```

This rule has two symbols on the right hand side, so two states, 6 and 3, are popped off of the stack, uncovering state 0. Consulting the description of state 0, looking for a goto on *sound*,

```
sound  goto 2
```

is obtained; thus state 2 is pushed onto the stack, becoming the current state.

In state 2, the next token, *DELL*, must be read. The action is "shift 5", so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the lookahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on *place*, the left side of rule 3, is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on *rhyme* causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained, indicated by "$end" in the *y.output* file. The action in state 1 when the endmarker is seen is to accept, successfully ending the parse.

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

5: Ambiguity and Conflicts

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

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

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

```
expr - expr - expr
```

the rule allows this input to be structured as either

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

or as

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

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

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

```
expr - expr - expr
```

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

```
expr - expr
```

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

```
— expr
```

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

Alternatively, when the parser has seen

```
expr — expr
```

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

```
expr — expr — expr
```

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

```
expr — expr
```

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

```
expr — expr
```

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

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

Yacc invokes two disambiguating rules by default:

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

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

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

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

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

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

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

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

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

can be structured according to these rules in two ways:

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

or

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

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

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

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

```
IF ( C1 ) stat
```

and then read the remaining input,

```
ELSE S2
```

and reduce

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

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

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

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

can be reduced by the if-else rule to get

```
IF ( C1 ) stat
```

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

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

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

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

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

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

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

state 23

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

ELSE    shift 45
        reduce 18
```

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

```
IF ( cond ) stat
```

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

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

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

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

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

6: Precedence

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

```
expr : expr OP expr
```

and

```
expr : UNARY expr
```

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

construct a parser that realizes the desired precedences and associativities.

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

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

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

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

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

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

%%

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

might be used to structure the input

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

as follows:

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

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

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

%%

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

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

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

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

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

7: Error Handling

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

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

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

legal. It then behaves as if the token "error" were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form

```
stat    :       error
```

would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement, and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general, but difficult to control. Somewhat easier are rules such as

```
stat    :       error ';'
```

Here, when there is an error, the parser attempts to skip over the statement, but will do so by skipping to the next ';'. All tokens after the error and before the next ';' cannot be shifted, and are discarded. When the ';' is seen, this rule will be reduced, and any "cleanup" action associated with it performed.

Another form of error rule arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error rule might be

```
input   :       error '\n' { printf( "Reenter last line: " ); } input
                        {       $$ = $4; }
```

There is one potential difficulty with this approach; the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example is better written

```
input   :       error '\n'
                        {       yyerrok;
                                printf( "Reenter last line: " );  }
                input
                        {       $$ = $4; }
        ;
```

As mentioned above, the token seen immediately after the "error" symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after error were to call some

sophisticated resynchronization routine, supplied by the user, that attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by yylex would presumably be the first token in a legal statement; the old, illegal token must be discarded, and the error state reset. This could be done by a rule like

```
stat    :       error
                        {       resynch();
                                yyerrok ;
                                yyclearin ;     }
        ;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors; moreover, the user can get control to deal with the error actions required by other portions of the program.

8: The Yacc Environment

When the user inputs a specification to Yacc, the output is a file of C programs, called *y.tab.c* on most systems (due to local file system conventions, the names may differ from installation to installation). The function produced by Yacc is called *yyparse*; it is an integer valued function. When it is called, it in turn repeatedly calls *yylex*, the lexical analyzer supplied by the user (see Section 3) to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) *yyparse* returns the value 1, or the lexical analyzer returns the endmarker token and the parser accepts. In this case, *yyparse* returns the value 0.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C program, a program called *main* must be defined, that eventually calls *yyparse*. In addition, a routine called *yyerror* prints a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using Yacc, a library has been provided with default versions of *main* and *yyerror*. The name of this library is system dependent; on many systems the library is accessed by a **−ly** argument to the loader. To show the triviality of these default programs, the source is given below:

```
main(){
        return( yyparse() );
        }
```

and

```
# include <stdio.h>

yyerror(s) char *s; {
        fprintf( stderr, "%s\n", s );
        }
```

The argument to *yyerror* is a string containing an error message, usually the string "syntax error". The average application will want to do better than this. Ordinarily, the program should keep track of the input line number, and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the lookahead token number at the time the error was detected; this may be of some interest in giving better diagnostics. Since the *main* program is probably supplied by the user (to read arguments, etc.) the Yacc library is useful only in small projects, or in the earliest stages of larger ones.

The external integer variable *yydebug* is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

9: Hints for Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following style hints owe much to Brian Kernighan.

a. Use all capital letters for token names, all lower case letters for nonterminal names. This rule comes under the heading of "knowing who to blame when things go wrong."

b. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.

c. Put all rules with the same left hand side together. Put the left hand side in only once, and let all following rules begin with a vertical bar.

d. Put a semicolon only after the last rule with a given left hand side, and put the semicolon on a separate line. This allows new rules to be easily added.

e. Indent rule bodies by two tab stops, and action bodies by three tab stops.

The example in Appendix A is written following this style, as are the examples in the text of this paper (where space permits). The user must make up his own mind about these stylistic questions; the central problem, however, is to make the rules visible through the morass of action code.

Left Recursion

The algorithm used by the Yacc parser encourages so called "left recursive" grammar rules: rules of the form

```
name    :       name rest_of_rule ;
```

These rules frequently arise when writing specifications of sequences and lists:

```
list    :       item
        |       list ',' item
        ;
```

and

```
seq     :       item
        |       seq item
        ;
```

In each of these cases, the first rule will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as

```
seq     :       item
        |       item seq
        ;
```

the parser would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion wherever reasonable.

It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
seq     :       /* empty */
        |       seq  item
        ;
```

Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if Yacc is asked to decide which empty sequence it has seen, when it hasn't seen enough to know!

Lexical Tie-ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names might be entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of 0 or more declarations, followed by 0 or more statements. Consider:

```
%{
        int dflag;
%}
  ... other declarations ...

%%

prog    :       decls  stats
        ;

decls   :       /* empty */
                        {       dflag = 1; }
        |       decls  declaration
        ;

stats   :       /* empty */
                        {       dflag = 0; }
        |       stats  statement
        ;

    ... other rules ...
```

The flag *dflag* is now 0 when reading statements, and 1 when reading declarations, *except for the first token in the first statement.* This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of "backdoor" approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Reserved Words

Some programming languages permit the user to use words like "if", which are normally reserved, as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of Yacc; it is difficult to pass information to the lexical analyzer telling it "this instance of 'if' is a keyword, and that instance is a variable". The user can make a stab at it, using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be *reserved*; that is, be forbidden for use as variable names. There are

powerful stylistic reasons for preferring this, anyway.

10: Advanced Topics

This section discusses a number of advanced features of Yacc.

Simulating Error and Accept in Actions

The parsing actions of error and accept can be simulated in an action by use of macros YYACCEPT and YYERROR. YYACCEPT causes *yyparse* to return the value 0; YYERROR causes the parser to behave as if the current input symbol had been a syntax error; *yyerror* is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple endmarkers or context-sensitive syntax checking.

Accessing Values in Enclosing Rules.

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit, but in this case the digit may be 0 or negative. Consider

```
sent    :       adj noun verb adj noun
                        { look at the sentence . . . }
        ;

adj     :       THE             {       $$ = THE; }
        |       YOUNG           {       $$ = YOUNG; }
        . . .
        ;

noun    :       DOG
                        {       $$ = DOG; }
        |       CRONE
                        {       if( $0 == YOUNG ){
                                        printf( "what?\n" );
                                        }
                                $$ = CRONE;
                                }
        ;
        . . .
```

In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG. Obviously, this is only possible when a great deal is known about what might precede the symbol *noun* in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism will save a great deal of trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. Yacc can also support values of other types, including structures. In addition, Yacc keeps track of the types, and inserts appropriate union member names so that the resulting parser will be strictly type checked. The Yacc value stack (see Section 4) is declared to be a *union* of the various types of values desired. The user declares the union, and associates union member names to each token and nonterminal symbol having a value. When the value is referenced through a $$ or $n construction, Yacc will automatically insert the appropriate union name, so that no unwanted conversions will take place. In addition, type checking commands such as *Lint*[5] will be far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union; this must be done by the user since other programs, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where Yacc can not easily determine the type.

To declare the union, the user includes in the declaration section:

```
%union {
        body of union ...
        }
```

This declares the Yacc value stack, and the external variables *yylval* and *yyval*, to have type equal to this union. If Yacc was invoked with the **-d** option, the union declaration is copied onto the *y.tab.h* file. Alternatively, the union may be declared in a header file, and a typedef used to define the variable YYSTYPE to represent this union. Thus, the header file might also have said:

```
typedef union {
        body of union ...
        } YYSTYPE;
```

The header file must be included in the declarations section, by use of %{ and %}.

Once YYSTYPE is defined, the union member names must be associated with the various terminal and nonterminal names. The construction

```
< name >
```

is used to indicate a union member name. If this follows one of the keywords %token, %left, %right, and %nonassoc, the union member name is associated with the tokens listed. Thus, saying

```
%left <optype> '+' '-'
```

will cause any reference to values returned by these two tokens to be tagged with the union member name *optype*. Another keyword, %type, is used similarly to associate union member names with nonterminals. Thus, one might say

```
%type <nodetype> expr stat
```

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no *a priori* type. Similarly, reference to left context values (such as $0 – see the previous subsection) leaves Yacc with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name, between < and >, immediately after the first $. An example of this usage is

```
rule    :       aaa { $<intval>$ = 3; } bbb
                        {       fun( $<intval>2, $<other>0 ); }
        ;
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in Appendix C. The facilities in this subsection are not triggered until they are used: in particular, the use of %type will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of $n or $$ to refer to something with no defined type is diagnosed. If these facilities are not triggered, the Yacc value stack is used to hold *int*'s, as was true historically.

11: Acknowledgements

Yacc owes much to a most stimulating collection of users, who have goaded me beyond my inclination, and frequently beyond my ability, in their endless search for "one more feature". Their irritating unwillingness to learn how to do things my way has usually led to my doing things their way; most of the time, they have been right. B. W. Kernighan, P. J. Plauger, S. I. Feldman, C. Imagna, M. E. Lesk, and A. Snyder will recognize some of their ideas in the current version of Yacc. C. B. Haley contributed to the error recovery algorithm. D. M. Ritchie, B. W. Kernighan, and M. O. Harris helped translate this document into English. Al Aho also deserves special credit for bringing the mountain to Mohammed, and other favors.

References

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey (1978).
2. A. V. Aho and S. C. Johnson, "LR Parsing," *Comp. Surveys* **6**(2) pp. 99-124 (June 1974).
3. A. V. Aho, S. C. Johnson, and J. D. Ullman, "Deterministic Parsing of Ambiguous Grammars," *Comm. Assoc. Comp. Mach.* **18**(8) pp. 441-452 (August 1975).
4. A. V. Aho and J. D. Ullman, *Principles of Compiler Design,* Addison-Wesley, Reading, Mass. (1977).
5. S. C. Johnson, "Lint, a C Program Checker," Comp. Sci. Tech. Rep. No. 65 (December 1977).
6. S. C. Johnson, "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages,* (January 1978).
7. B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," *Comm. Assoc. Comp. Mach.* **18** pp. 151-157 (March 1975).
8. M. E. Lesk, "Lex — A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey (October 1975).

Appendix A: A Simple Example

This example gives the complete Yacc specification for a small desk calculator; the desk calculator has 26 registers, labeled "a" through "z", and accepts arithmetic expressions made up of the operators +, −, *, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression at the top level is an assignment, the value is not printed; otherwise it is. As in C, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a Yacc specification, the desk calculator does a reasonable job of showing how precedences and ambiguities are used, and demonstrating simple error recovery. The major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately, line by line. Note the way that decimal and octal integers are read in by the grammar rules; This job is probably better done by the lexical analyzer.

```
%{
# include <stdio.h>
# include <ctype.h>

int regs[26];
int base;

%}

%start list

%token DIGIT LETTER

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS    /* supplies precedence for unary minus */

%%    /* beginning of rules section */

list    :       /* empty */
        |       list stat '\n'
        |       list error '\n'
                        {       yyerrok; }
        ;

stat    :       expr
                        {       printf( "%d\n", $1 ); }
        |       LETTER '=' expr
                        {       regs[$1] = $3; }
        ;

expr    :       '(' expr ')'
                        {       $$ = $2; }
        |       expr '+' expr
                        {       $$ = $1 + $3; }
        |       expr '-' expr
                        {       $$ = $1 - $3; }
```

```
        |       expr '*' expr
                        {       $$ = $1 * $3; }
        |       expr '/' expr
                        {       $$ = $1 / $3; }
        |       expr '%' expr
                        {       $$ = $1 % $3; }
        |       expr '&' expr
                        {       $$ = $1 & $3; }
        |       expr '|' expr
                        {       $$ = $1 | $3; }
        |       '-' expr        %prec UMINUS
                        {       $$ = - $2; }
        |       LETTER
                        {       $$ = regs[$1]; }
        |       number
        ;

number:         DIGIT
                        {       $$ = $1;   base = ($1==0) ? 8 : 10; }
        |       number DIGIT
                        {       $$ = base * $1 + $2; }
        ;

%%      /* start of programs */

yylex() {               /* lexical analysis routine */
          /* returns LETTER for a lower case letter, yylval = 0 through 25 */
          /* return DIGIT for a digit, yylval = 0 through 9 */
          /* all other characters are returned immediately */

        int c;

        while( (c=getchar()) == ' ' ) {/* skip blanks */ }

        /* c is now nonblank */

        if( islower( c ) ) {
                yylval = c - 'a';
                return ( LETTER );
                }
        if( isdigit( c ) ) {
                yylval = c - '0';
                return( DIGIT );
                }
        return( c );
        }
```

Appendix B: Yacc Input Syntax

This Appendix has a description of the Yacc input syntax, as a Yacc specification. Context dependencies, etc., are not considered. Ironically, the Yacc input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule, immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier, and decide whether the next token (skipping blanks, newlines, comments, etc.) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERS, but never as part of C_IDENTIFIERs.

```
        /* grammar for the input to Yacc */

        /* basic entities */
%token  IDENTIFIER      /*  includes identifiers  and literals */
%token  C_IDENTIFIER    /*   identifier (but not literal) followed by colon   */
%token  NUMBER                  /*   [0-9]+   */

        /* reserved words:  %type => TYPE, %left => LEFT, etc. */

%token  LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token  MARK  /* the %% mark */
%token  LCURL /* the %{ mark */
%token  RCURL /* the %} mark */

        /* ascii character literals stand for themselves */

%start  spec

%%

spec    :       defs MARK rules tail
        ;

tail    :       MARK { In this action, eat up the rest of the file }
        |       /* empty: the second MARK is optional */
        ;

defs    :       /* empty */
        |       defs def
        ;

def     :       START IDENTIFIER
        |       UNION { Copy union definition to output }
        |       LCURL { Copy C code to output file } RCURL
        |       ndefs rword tag nlist
        ;

rword   :       TOKEN
        |       LEFT
        |       RIGHT
```

```
        |       NONASSOC
        |       TYPE
        ;

tag     :       /* empty: union tag is optional */
        |       '<' IDENTIFIER '>'
        ;

nlist   :       nmno
        |       nlist nmno
        |       nlist ',' nmno
        ;

nmno    :       IDENTIFIER              /* NOTE: literal illegal with %type */
        |       IDENTIFIER NUMBER       /* NOTE: illegal with %type */
        ;

        /* rules section */

rules   :       C_IDENTIFIER rbody prec
        |       rules rule
        ;

rule    :       C_IDENTIFIER rbody prec
        |       '|' rbody prec
        ;

rbody   :       /* empty */
        |       rbody IDENTIFIER
        |       rbody act
        ;

act     :       '{' { Copy action, translate $$, etc. } '}'
        ;

prec    :       /* empty */
        |       PREC IDENTIFIER
        |       PREC IDENTIFIER act
        |       prec ';'
        ;
```

Appendix C: An Advanced Example

This Appendix gives an example of a grammar using some of the advanced features discussed in Section 10. The desk calculator example in Appendix A is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants, the arithmetic operations +, −, *, /, unary −, and = (assignment), and has 26 floating point variables, "a" through "z". Moreover, it also understands *intervals*, written

(x , y)

where *x* is less than or equal to *y*. There are 26 interval valued variables "A" through "Z" that may also be used. The usage is similar to that in Appendix A; assignments return no value, and print nothing, while expressions print the (floating or interval) value.

This example explores a number of interesting features of Yacc and C. Intervals are represented by a structure, consisting of the left and right endpoint values, stored as *double*'s. This structure is given a type name, INTERVAL, by using *typedef*. The Yacc value stack can also contain floating point scalars, and integers (used to index into the arrays holding the variable values). Notice that this entire strategy depends strongly on being able to assign structures and unions in C. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of YYERROR to handle error conditions: division by an interval containing 0, and an interval presented in the wrong order. In effect, the error recovery mechanism of Yacc is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (e.g. scalar or interval) of intermediate expressions. Note that a scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through Yacc: 18 Shift/Reduce and 26 Reduce/Reduce. The problem can be seen by looking at the two input lines:

2.5 + (3.5 − 4.)

and

2.5 + (3.5 , 4.)

Notice that the 2.5 is to be used in an interval valued expression in the second example, but this fact is not known until the "," is read; by this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator: one when the left operand is a scalar, and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflicts will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive, but not very general. If there were many kinds of expression types, instead of just two, the number of rules needed would increase dramatically, and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value, and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C library routine *atof* is used to do the actual conversion from a character string to a double precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar, provoking a syntax error in the parser, and thence error recovery.

```
%{

# include <stdio.h>
# include <ctype.h>

typedef struct interval {
        double lo, hi;
        } INTERVAL;

INTERVAL vmul(), vdiv();

double atof();

double dreg[ 26 ];
INTERVAL vreg[ 26 ];

%}

%start  lines

%union  {
        int ival;
        double dval;
        INTERVAL vval;
        }

%token <ival> DREG VREG         /* indices into dreg, vreg arrays */

%token <dval> CONST             /* floating point constant */

%type <dval> dexp               /* expression */

%type <vval> vexp               /* interval expression */

        /* precedence information about the operators */

%left   '+' '-'
%left   '*' '/'
%left   UMINUS      /* precedence for unary minus */

%%

lines   :       /* empty */
        |       lines line
        ;

line    :       dexp '\n'
                        {       printf( "%15.8f\n", $1 ); }
        |       vexp '\n'
                        {       printf( "(%15.8f , %15.8f )\n", $1.lo, $1.hi ); }
        |       DREG '=' dexp '\n'
                        {       dreg[$1] = $3; }
        |       VREG '=' vexp '\n'
```

```
                { vreg[$1] = $3; }
        |   error '\n'
                { yyerrok; }
        ;

dexp    :   CONST
        |   DREG
                { $$ = dreg[$1]; }
        |   dexp '+' dexp
                { $$ = $1 + $3; }
        |   dexp '-' dexp
                { $$ = $1 - $3; }
        |   dexp '*' dexp
                { $$ = $1 * $3; }
        |   dexp '/' dexp
                { $$ = $1 / $3; }
        |   '-' dexp    %prec UMINUS
                { $$ = - $2; }
        |   '(' dexp ')'
                { $$ = $2; }
        ;

vexp    :   dexp
                { $$.hi = $$.lo = $1; }
        |   '(' dexp ',' dexp ')'
                {
                $$.lo = $2;
                $$.hi = $4;
                if( $$.lo > $$.hi ){
                        printf( "interval out of order\n" );
                        YYERROR;
                        }
                }
        |   VREG
                { $$ = vreg[$1]; }
        |   vexp '+' vexp
                { $$.hi = $1.hi + $3.hi;
                  $$.lo = $1.lo + $3.lo; }
        |   dexp '+' vexp
                { $$.hi = $1 + $3.hi;
                  $$.lo = $1 + $3.lo; }
        |   vexp '-' vexp
                { $$.hi = $1.hi - $3.lo;
                  $$.lo = $1.lo - $3.hi; }
        |   dexp '-' vexp
                { $$.hi = $1 - $3.lo;
                  $$.lo = $1 - $3.hi; }
        |   vexp '*' vexp
                { $$ = vmul( $1.lo, $1.hi, $3 ); }
        |   dexp '*' vexp
                { $$ = vmul( $1, $1, $3 ); }
        |   vexp '/' vexp
                { if( dcheck( $3 ) ) YYERROR;
                  $$ = vdiv( $1.lo, $1.hi, $3 ); }
```

```
        |       dexp '/' vexp
                        {       if( dcheck( $3 ) ) YYERROR;
                                $$ = vdiv( $1, $1, $3 ); }
        |       '-' vexp        %prec UMINUS
                        {       $$.hi = -$2.lo;  $$.lo = -$2.hi;  }
        |       '(' vexp ')'
                        {       $$ = $2; }
        ;

%%

# define BSZ 50      /* buffer size for floating point numbers */

        /* lexical analysis */

yylex(){
        register c;

        while( (c=getchar()) == ' ' ){ /* skip over blanks */ }

        if( isupper( c ) ){
                yylval.ival = c - 'A';
                return( VREG );
                }
        if( islower( c ) ){
                yylval.ival = c - 'a';
                return( DREG );
                }

        if( isdigit( c ) || c=='.' ){
                /* gobble up digits, points, exponents */

                char buf[BSZ+1], *cp = buf;
                int dot = 0, exp = 0;

                for( ; (cp-buf)<BSZ ; ++cp,c=getchar() ){

                        *cp = c;
                        if( isdigit( c ) ) continue;
                        if( c == '.' ){
                                if( dot++ || exp ) return( '.' );   /* will cause syntax error */
                                continue;
                                }

                        if( c == 'e' ){
                                if( exp++ ) return( 'e' );   /* will cause syntax error */
                                continue;
                                }

                        /* end of number */
                        break;
                        }
                *cp = '\0';
                if( (cp-buf) >= BSZ ) printf( "constant too long: truncated\n" );
```

```
            else ungetc( c, stdin );  /* push back last char read */
            yylval.dval = atof( buf );
            return( CONST );
            }
        return( c );
        }

INTERVAL hilo( a, b, c, d ) double a, b, c, d; {
        /* returns the smallest interval containing a, b, c, and d */
        /* used by *, / routines */
        INTERVAL v;

        if( a>b ) { v.hi = a;   v.lo = b; }
        else { v.hi = b;   v.lo = a; }

        if( c>d ) {
                if( c>v.hi ) v.hi = c;
                if( d<v.lo ) v.lo = d;
                }
        else {
                if( d>v.hi ) v.hi = d;
                if( c<v.lo ) v.lo = c;
                }
        return( v );
        }

INTERVAL vmul( a, b, v ) double a, b;  INTERVAL v; {
        return( hilo( a*v.hi, a*v.lo, b*v.hi, b*v.lo ) );
        }

dcheck( v ) INTERVAL v; {
        if( v.hi >= 0. && v.lo <= 0. ){
                printf( "divisor interval contains 0.\n" );
                return( 1 );
                }
        return( 0 );
        }

INTERVAL vdiv( a, b, v ) double a, b;  INTERVAL v; {
        return( hilo( a/v.hi, a/v.lo, b/v.hi, b/v.lo ) );
        }
```

Appendix D: Old Features Supported but not Encouraged

This Appendix mentions synonyms and features which are supported for historical continuity, but, for various reasons, are not encouraged.

1. Literals may also be delimited by double quotes ‘“”’.
2. Literals may be more than one character long. If all the characters are alphabetic, numeric, or _, the type number of the literal is defined, just as if the literal did not have the quotes around it. Otherwise, it is difficult to find the value for such literals.

   The use of multi-character literals is likely to mislead those unfamiliar with Yacc, since it suggests that Yacc is doing a job which must be actually done by the lexical analyzer.
3. Most places where % is legal, backslash “\” may be used. In particular, \\ is the same as %%, \left the same as %left, etc.
4. There are a number of other synonyms:

   %< is the same as %left
   %> is the same as %right
   %binary and %2 are the same as %nonassoc
   %0 and %term are the same as %token
   %= is the same as %prec

5. Actions may also have the form

   ={ . . . }

   and the curly braces can be dropped if the action is a single C statement.
6. C code between %{ and %} used to be permitted at the head of the rules section, as well as in the declaration section.

Plexus Release 1.0/3.0 of the UNIX Virtual Protocol Machine

This document draws heavily from the memorandum on VPM supplied in the *Programmer's Manual for UNIX System III, Volume 2B* (October 1981).

Changes have been made to reflect the Plexus implementation of VPM.

Plexus release 1.0 of VPM is for Z8000-based Plexus computer systems (P/25 and P/40); release 3.0 is for MC68000-based Plexus computer systems (P/35, P/60, P/65, and P/75).

ABSTRACT

This document describes the initial release of the Virtual Protocol Machine (VPM), a UNIX* synchronous communication subsystem, as implemented on a Plexus computer system. The Plexus release of VPM is built around the Intelligent Communications Processor (ICP), a Z8000 based microcomputer that connects to the MULTIBUS of a Plexus computer. The VPM is a software construct for implementing link protocols on the ICP 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 ICP and interprets the virtual machine instruction set. A UNIX driver 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 implementaton 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 ICP program that controls the overall operation of the ICP and interprets the protocol script.
3. *si.c*: the UNIX driver that provides the interface to the VPM. This source is not available for distribution.
4. *vpmstart*: a UNIX command that copies a load module into the ICP 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.

---

* UNIX is a trademark of Bell Laboratories.

INTRODUCTION

The Virtual Protocol Machine (VPM) is a UNIX synchronous communications subsystem built around the ICP microcomputer.

The VPM is a software construct for implementing link protocols on the ICP 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 ICP and interprets the virtual machine instruction set. This program also controls the communications line(s) and provides the interface to the UNIX host machine. The compiled protocol script is loaded with the interpreter into the ICP. A UNIX driver provides the interface between the user process's *open*, *close*, *read*, and *write* calls and the protocol script executed by the interpreter in the ICP.

Besides providing the benefits of a high-level language implementations 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 ICP, they are the same regardless of the software running in the main CPU or, for that matter, regardless of the CPU itself. 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. Another possibility for portability 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.

PLEXUS RELEASE 1.0/3.0

Plexus release 1.0 of VPM is restricted to byte-oriented half-duplex protocols such as BISYNC. Plexus release 3.0 includes bit-oriented, full-duplex protocols such as HDLC.

This release of the VPM software is intended for use with UNIX Sys3. Operation with other versions of UNIX has not been tested. The VPM software consists of five components:

1. *vpmc*: a UNIX compiler for the protocol description language.
2. VPM interpreter: the ICP program that controls the overall operation of the ICP and interprets the protocol script.

---

1. B. W. Kernighan, *RATFOR - A Preprocessor for a Rational Fortran*, Bell Laboratories.
2. B. W. Kernighan, *The M4 Macro Processor*, Bell Laboratories.

3. *si.c*: the UNIX driver that provides the interface to the VPM. This source is not available for distribution.
4. *vpmstart*: a UNIX command that copies a load module into the ICP and starts it.
5. *vpmtrace*: a UNIX command that prints an event trace for debugging while the protocol is running.

A release tape containing the VPM software is available from Plexus Computers. Installation procedures are described in the *Plexus VPM Rev. 3.0 Release Notice*, Plexus publication number 98-40058.

Plexus Release 2.0 of the UNIX Virtual Protocol Machine

The following document is based on the Bell Laboratories memorandum and adds appropriate changes for the Plexus implementation of VPM.

The Bell UNIX documentation refers to Release 1.0 and 2.0 of VPM. This maps to the Plexus versions as follows:

Bell	Plexus
VPM 1.0	Plexus VPM 1.0 for Z8000-based computers (P/25, P/40) Plexus VPM 3.0 for MC68000-based computers (P/35, P/60, P/65, P/75)
VPM 2.0	Plexus VPM 3.0 for MC68000 based systems only (P/35, P/60, P/65, P/75)

ABSTRACT

This document 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 Intelligent Communications Processor (ICP) in a high-level language. The VPM software consists of a protocol compiler, a UNIX driver, an interpreter that executes in the ICP, 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; and
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 Plexus Sys3 UNIX 3.2 system and testing it to ensure proper operation are given.

Introduction

This document describes the second release of the UNIX* Virtual Protocol Machine (VPM). The first release was described in the publication *Plexus VPM Release 1.0*, which should be read as background for this memorandum.

VPM is a general-purpose UNIX interface for synchronous communications lines. VPM allows link-level protocols such as BISYNC and HDLC to be inplemented on the Plexus ICP microcomputer in a high-level language. The hardware required to support VPM is a Plexus host computer, and an ICP. The link-level communications protocol is excuted by the VPM interpreter running in the ICP Plexus ICP. 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 Plexus ICP, 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 ICP. The load module contains the VPM interpreter and a compiled representation of the protocol. The 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 of 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 which 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, and
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 attached manual entry for *vpmc*(1C). An overview of these primitives follows.

---

* UNIX is a trademark of AT&T Bell Laboratories

The VPM interpreter maintains a set of queues for transmit buffers. When a transmit buffer is passed to the ICP by the UNIX driver, the buffer is appended to the unopened-transmit-buffer queue. The protocol script in the ICP 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 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 ICP 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 rceived, 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

Since the VPM interpreter and a protocol script generally use most of the memory of the ICP 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 ICP.

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 ICP. Their capabilities include a means to queue transmit and empty receive buffers for use by the protocol script in the ICP, 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 I 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 lease two other protocol modules have been written thus far. They are the Synchronous Terminal Interface [4, *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 ICP and one interface-module minor device* is 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 since no more than four ICPs are supported. In general, each protocol module can have up to 256 minor devices. The VPM Release 2.0 protocol module, however, can have at most l6 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 ICP minor device. The low-order four bits of the protocol-module minor devie number determine the protocol-module minor device; the next two bits determine the interface-module minor device; the next two bits determine the ICP 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 l6 bits of buffer address */
    char  d_hbits;           /* High-order 2 bits of buffer address  */
    char  d_sta;             /* Protocol-dependent */
    char  d_type;            /* 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 l8-bit MULTIBUS-mapped buffer address; the low-order l6 bits must be in *d_adres* and the high-order two bits must be in the low-order two bits of *id_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 *vpmc*(1C). *D_buf* contains a pointer to a system buffer descripter; 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 l6 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 *vmpsnap* commands are available for reading and formatting the data passed via these two minor devices. These two commands are described in the attached manual entry for *vpmstart*(IC). 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 ICP 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 ICP 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 *time* 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 it 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 the bit-oriented (HDLC) interpreter only.

Appendix 5 contains detailed instructions for adding VPM Release 2.0 to a Plexus Sys3 UNIX 3.2 system. Appendix 6 decribes 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.

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 ICP. The ICP 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 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 ICP 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 since 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 vpmdb *bdp;

This function is used to pass an empty receive buffer for use by the interpreter in the ICP. *Bdp* is a pointer to a buffer descriptor or null. If *bdp* is not a null pointer, the buffer decriptor 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 ICP. 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.

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 a 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 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 ICP. 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_sequn;         /* 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 ICP identified by *dev*. *Type* is a number that identifies the program running in the ICP and must agree with the value specified when the ICP load module was loaded into the ICP. 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 *fpmstop* has not been called, or if the ICP is not running or was loaded using a different magic number, EACCESS 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 ICP 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 opens 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 driver is opened for both reading only or for writing only, the *open* is non-exclusive and subsequent *opens* 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 *rtnbuf* (BISNYC) or *rtnfrm* (HDLC) primitive, which causes the buffer descriptor pointer to be passed to the interface module's 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 *inode* 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 therfore 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 *rtnbuf* (BISYNC) or a *rtnfrm* (HDLC).
>
> RRTNEBUF - *Bdf* 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 attached manual entry for *vmp*(4).

The values for RRTNXBUF, RRTNRBUF, RRTNEBUF, and ERRITERM are defined in the */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 attached 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 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:

> VPMTRO - 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 l5. 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 l00 l5 0

Called if *vpmstart* returns an error code. The first field of the printed record contains 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('0', dev, vp->vt_state, 0)

244 o l00 l 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 l00 13 0

Called from *vpmclose* just before the state bits are initialized.

vpmsave ('w', dev, ct, dp)

246 w l00 l000

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, ent, dp->d_bos)

247 r l00 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->vbstate, 0)

248 s l00 40l 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 l00 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 l00 l 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 l00 l 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 l00 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 assembley-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 attached manual entry for *vpm*(4).

vpmsave ('P', dev, sel4, sel6)

254 P l00 2105 l055

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 l00 3 0

Called form *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 l00 40l 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->xbkmc)

257 C l00 l 0

Called from *vpmclean* just before the data structure associated with *dev* is initialized.

Appendix 5

Adding VPM to a UNIX Release 3.0 System

The UNIX Releae 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 b used to make and install all VPM commands.

To add the VPM and trace drivers to an 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   l5   l6   4
trace   0   35   206   tr    0   0   l6   4    l
```

2. Add the following line to the file **/usr/src/uts/*/cf/cfigpa** (or its equivalent):

   ```
   vpm   0   0   0   0
   ```

   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   0
   ```

   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 ICPs are being added to the system, add the following line to the same file for each ICP:

   ```
   icp  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 ICP, VPM, and trace device. To make these special files, use *mknod*(1M) as follows:

For ICPs:

/etc/mknod /dev/ic? c X ?

where X is the major device number of the ICP driver as printed by *config* -t (see the manual entry for *config*(1M)[4] and ? is the minor device number, which 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 whose binary representation is defined as follows: the low-order four bits specify one of up to l6 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 ICP 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.

Appendix 6

Testing VPM

During the course of developing and testing VPM, a number of programs and test procedures have evolved that 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*.

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 loopback script such as *loop.r* has been loaded into the ICP. 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 ICP:

   /etc/vpmstart /dev/icp? 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 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 to 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/vpmO*. 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 ICP, refer to the *Plexus Sys3 3.2 Release Notice* (98-40081).

Testing a script requires two ICPs, which may be on different host computers. The ICPs must be connected by a pair of full-duplex synchronous modems or by a full-duplex synchronous null modem*. *Sr* should be executed simultaneously on both machines to read and write the VPM device associated with each ICP. If both ICPs 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 ICPs.

To obtain maximum efficiency from a 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.

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. Compile *itr.r* and load the resulting *a.out* into the ICP using the procedure described above for *lapb.r*, substituting *itr* for *lapb*. A loop-back test can then be run using *tset* or *sr*.

---

* A suitable null modem is the Avanti 300. which is manufactured by Avanti Communications CorPoration. NewPort, RI.

Setting Up UNIX

R. C. Haight
T. J. Kowalski
M. J. Petrella
L. A. Wehr

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

1.1 Prerequisites

Before attempting to generate a UNIX† system, you should understand that a considerable knowledge of the related documentation is required and assumed. In particular, you should be very familiar with the following documents:

- *The UNIX Time-Sharing System*
- *UNIX User's Manual*
- *C Reference Manual*
- *Administrative Advice for UNIX*

A complete set of pertinent documentation is contained in *Documents for UNIX*. Throughout this document, each reference of the form *name*(N), where N is a Arabic number, refers to entry *name* in Section N of the *UNIX User's Manual*.

You must have a basic understanding of the operation of the hardware. This includes the console panel, the tape drives, and the disk drives, all of which are assumed to have standard UNIX addresses and interrupt vectors. It is also assumed that the hardware works and has been completely installed. All appropriate DEC diagnostics and the Equipment Test Package should have been run to test the configuration, and you must have a detailed description of the hardware, including device addresses, interrupt vectors, and bus levels. This information is necessary to generate the UNIX system.

See Attachment 1 for a list of supported CPUs and devices.

1.2 Procedure

UNIX is distributed on a single, multi-file magnetic tape, recorded in 9-track format at 800 bpi. Distribution tapes will be marked either "PDP-11" or "VAX"; be sure you have the correct tape for your machine.

The initial load program will copy a file system from tape (VAX: TE16; PDP-11: either a TU10 or a TU16) to disk (VAX: RP06; PDP-11: either an RP03, an RK05, an RL01, or an RP06). In this document, we consider RP04 and RP05 drives to be equivalent to RP06 drives; any differences will be noted explicitly. Once the *root* file system has been successfully loaded to disk, UNIX may be booted and the available utility programs may then be used to complete the installation.

The remaining files on the tape contain source text and supplemental commands. These files contain essential information to generate a new system that will match your particular hardware and software environment.

In order for any of the update procedures to work correctly, you *must* be *running* UNIX/TS or PWB/UNIX Edition 2.0. Older versions of UNIX cannot be correctly *updated* with a UNIX system. The *cpio*(1) program will not replace any file if its replacement has a modification time that is

---

† UNIX is a Trademark of AT&T Bell Laboratories.

less than (i.e., earlier than) the modification time of the original file. This can be due to local modifications. Furthermore, certain administrative files (e.g., */etc/passwd*, */usr/lib/crontab*) are sent with a modification time of January 1, 1970 to ensure that they do not replace their counterparts during updates. Any file not copied will cause *cpio*(1) to print a message to that effect. These messages should always be investigated to ensure that any files not copied were of that type. However, note that, depending on respective modification times, a locally-modified file may get updated, thus destroying the local modifications.

There are several difficulties that can arise when installing a UNIX system. One of the most common problems is running out of disk space when performing an update. Should this occur, the original contents of the file system should be restored from a backup copy and the contents of the update tape should be read into a spare file system using the *cpio*(1) program. Unwanted material can then be removed and the original file system can be updated from this new file system using the *-p* option of *cpio*(1). Modification times of files should also be preserved using the *-m* option of *cpio*(1).

This document is not strictly linear. Read it thoroughly, from start to finish, and then read it again. Also, remove the write-protect ring, if present, from the distribution tape to guard against accidental erasure.

2. LOAD PROCEDURES

2.1 Distribution Tape Format

The eight files are: a loader, a physical copy of the *root* file system, the *cpio*(1) program, a *cpio*(1) structured copy of the *root* file system, and three files (*cpio*) that make up */usr*. *Root* refers to the directory "/", which is the root of all the directory trees. The format of this tape is as follows:

record 0:	Tape boot loader-512-bytes;
record 1:	Tape boot loader-512-bytes;
remainder of file 1:	Initial load program-several 512-byte records;
end-of-file	
file 2:	*root* file system (physical)-several 5,120-byte records (blocking factor 10);
end-of-file	
file 3:	*cpio* program (latest version)-several 512-byte records;
end-of-file	
file 4:	*root* file system (structured in *cpio* format)-several 5,120-byte records (to be used *only* for updating an earlier UNIX);
end-of-file	
file 5:	*/usr* file system except *src* and *man* (*cpio*).
end-of-file	
file 6:	*/usr/src* part 1.
end-of-file	
file 7:	*/usr/src* part 2.
end-of-file	
file 8:	*/usr/man*.
end-of-file	

The *root* (/) file system contains the following directories:

bck:	Directory used to mount a backup file system for file restoring.
bin:	Public commands; most of what is described in Section 1 of the *UNIX User's Manual*.
dev:	Special files, all the devices on the system.

etc:	Administrative programs and tables.
lib:	Public libraries, parts of the assembler, C compiler.
mnt:	Directory used to mount a file system.
lost+found:	Directory used by *fsck*(1M) for disconnected files.
stand:	Stand-alone programs.
tmp:	Directory used for temporary files; should be cleaned at reboot.
usr:	Directory used to mount the */usr* file system; user directories often kept here also.

2.2 Initial Load of root

Mount the distribution tape on drive 0 and position it at the load point.

2.2.1 PDP-11

Bootstrap the tape by reading either record 0 or record 1 into memory starting at address 0 and start execution at address 0. This may be accomplished by using a standard DEC ROM bootstrap loader, a special ROM, or some manual procedure; see *romboot*(8), *tapeboot*(8), and *70boot*(8).

2.2.2 VAX

See "Installation Boot Procedures" under *vaxops*(8). This *UNIX User's Manual* entry describes initial tape booting, and modification of the console floppy disk to simplify UNIX administration.

2.3 Common to PDP-11 and VAX

The tape boot loader will then type "UNIX tape boot loader" on the console terminal and read in and execute the initial load program. The program will then type detailed instructions about the operation of the program on the console terminal. First, it will ask what type of CPU you have. Second, the program will ask what type of disk drive you have and which drive you plan to use for the copy. The disk controller used must be at the standard DEC address indicated by the program. However, other disk controllers on your system may be at non-standard addresses. You must mount a formatted, ECC flag-free pack on the drive you have indicated. If necessary, use the appropriate DEC diagnostic program to format the pack. Note that the pack will be written on. Third, the program will ask what type of tape drive you have and which drive contains the tape. Normally, this will be drive 0, but the program will work with other drives. Note that the tape is currently positioned correctly after the end-of-file between the initial load program and the *root* file system. When everything is ready, the program will copy the file system from the tape to disk and give instructions for booting UNIX. After the copy is complete and you have booted the basic version of UNIX, check (using *fsck*(1M)) the *root* file system and browse through it.

2.3.1 PDP-11/70 Only

The file */stand/mmtest* is a stand-alone memory mapping diagnostic program for the PDP-11/70. 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 CPU.

2.4 Update of root

It is very important that the system be running in *single-user* mode during the update phase. To update an already existing *root* file system, files three and four will be used. It is necessary to first make a copy of your *root* file system using *volcopy*(1M) and then update this copy. The copy should be made on a separate disk pack using the same section number as your *root* file system (always section 0). Also, after the update is completed, check if any of your local administrative files in the directory */etc* need modification. Most of these are mentioned in Section 4 below.

Mount the tape on drive 0 and position it at the load point. We assume that disk drive 1 is available for making the copy, and that the *root* file system is on */dev/rp0*. This shell procedure will be different for RK05 and RL01 disk drives. The following procedure will first make a copy of the *root* file system, and then update this copy. Note that */dev/mt4* refers to tape drive 0 but has the side effect of spacing forward to the next end-of-file (no rewind option). The *B* option of *cpio* specifies that input is in 5,120-byte records:

```
volcopy root /dev/rrp0 pkname1 /dev/rrp10 pkname2
mount /dev/rp10 /bck
#   The 2 echoes will move the tape to file 3
echo </dev/mt4
echo </dev/mt4
cp /dev/mt4 /bck/bin/cpio
chmod 755 /bck/bin/cpio
chown bin /bck/bin/cpio
cd /bck
/bck/bin/cpio -idmB </dev/rmt0
cd /
umount /dev/rp10
```

Pkname1 and *pkname2* are the volume names of the source and destination disk packs, respectively. If the new copy is satisfactory, shut down and halt the system, change disk packs, and reboot the system using the new *root*.

2.5 Files 5, 6, 7, and 8 (/usr) Format

File 5 contains the */usr* file system in *cpio*(1) format (5,120-byte records). The */usr* file system contains commands and files that must be available (mounted) when the system is in *multi-user* mode. The tape contains the following directories:

adm:	Miscellaneous administrative command and data files, including the connect-time accounting file *wtmp* and the process accounting file *pacct*.
bin:	Public commands; an overflow for */bin*.
dict:	Dictionaries for word processing programs.
games:	Various demonstration and instructional programs.
include:	Public C language *#include* files.
lib:	Archive libraries, including the text processing macros; also contains data files for various programs, such as *spell*(1) and *cron*(1M).
mail:	Mail directory.
man:	Source for the *UNIX User's Manual*; see *man*(1).
lost+found:	Directory used by *fsck*(1M) for disconnected files.
news:	Place for all the various system news.
pub:	Handy public information, e.g., table of ASCII characters.
spool:	Spool directory for daemons.
src:	Source for commands, libraries, the system, etc.
tmp:	Directory for temporary files; should be cleaned at reboot.

A table of contents of this tape may be obtained by using the *cpio*(1) options *-itB*. Also, after installation, files and directories deemed useless by the local administrator may be easily removed. Alternately, only parts of the tape may be extracted using the pattern matching capabilities of *cpio*(1).

2.6 Initial Load of /usr

Mount a file system (device) as */usr*. The ultimate size and location of this file system on a device is an administrative decision; initially, the following procedure will suffice:

```
#   The 4 echoes will move the tape to file 5
#   (mt4 is the no-rewind device)
echo </dev/mt4
echo </dev/mt4
echo </dev/mt4
echo </dev/mt4
cd /
mkfs /dev/rrp1 35000 7 418
#   The magic numbers "7 418" above refer to free-list ordering:
#   (rotational angle of 7 and 418 blocks/cylinder for RP04/5/6/s)
#   If you have RL01 drives, use mkfs /dev/rrl1 10240 13 20
#   If you have RK05 drives, use mkfs /dev/rrk1 4872 3 24
labelit /dev/rrp1 usr pkname
mount /dev/rp1 /usr
cd /usr
cpio -idmB </dev/rmt4  #  file 5: /usr except /src and /man
cd /usr/src
cpio -idmB </dev/rmt4  #  file 6: 1st part of /usr/src
cpio -idmB </dev/rmt4  #  file 7: last of /usr/src
cd /usr/man
cpio -idmB </dev/rmt0  #  file 8: /usr/man
#   If you have RL01 or RK05 drives, you will have to use separate
#   packs for files 5-8.
```

Pkname is the volume name of the pack (e.g., "p0001").

Because */usr* must be mounted when the system is in *multi-user* mode, the file */etc/rc* must be changed to include the command lines to mount and unmount the file systems in *single-user* and *multi-user* mode. These lines must be inserted at the appropriate places in */etc/rc*, as indicated by comments in the prototype file. Next the file */etc/checklist* should be changed to include */dev/rrp1*, */dev/rrl1*, or */dev/rrk1*; see also *fsck*(1M), *labelit*(1M), *mkfs*(1M), *mount*(1M), and *checklist*(5).

2.7 Update of /usr

It is advisable that the system be running in *single-user* mode during the update phase. It is also wise to first make a copy of your */usr* file system for backup purposes. Next, mount the distribution tape on drive 0 and position it at file 5. The */usr* file system *must* also be mounted. The following procedure will perform the update:

```
cd /usr
cpio -idmB </dev/rmt4
cpio -idmB </dev/rmt4
cpio -idmB </dev/rmt4
cpio -idmB </dev/rmt0
```

3. CONFIGURATION PLANNING

3.1 UNIX Configuration

The basic UNIX operating system supplied supports only the console, a disk controller (disk drive 0), and a tape controller (tape drive 0). The actual configuration of your system must be described by you. All of the UNIX operating system source code and object libraries are in */usr/src/uts*. All of the configuration information is kept in the directory */usr/src/uts/*/cf*. There are only two files that must be changed to reflect your system configuration, *low.s* and *conf.c*; the program *config*(1M) should be used to make these changes.

Config requires a *system description file* and produces the two needed files. The first part of the system description file lists all of the hardware devices on your system. Next, various system information is listed. A brief explanation of this information follows; for more details of syntax and structure, see *config*(1M) and the associated *master*(5); TABLE I lists the values and sizes of the various parameters for the different CPUs.

TABLE I. UNIX Configuration Parameters

Item	PDP-11/34, /23		PDP-11/45, /70		VAX-11/780	
	Value	*Size*	*Value*	*Size*	*Value*	*Size*
nswap	1000	-	3000	-	9000	-
buffers	15-20	26†	25-60	26†	50-120‡	560
sabufs	4-6	538	10-15	538	-	-
inodes	30-50	74	100-250	74	100-250	76
files	30-50	8	100-250	8	100-250	12
mounts	3-5	8	8-16	8	8-16	16
coremap	50-100	4	50-100	4	-	-
swapmap	50-100	4	50-100	4	50-100	4
calls	15-30	6	30-60	6	30-60	12
procs	30-50	30	50-200	30	50-200	52
texts	10-15	12	25-50	12	25-50	16
clists	10-20	28	100-300	28	100-300	32
maxup	15	-	15	-	25	-

† Plus 512 bytes outside system space.
‡ 127 buffers is the system maximum.

- *root*-Specifies the device where the *root* file system is to be found. The device must be a block device with read/write capability because this device will be mounted read/write as "/". Thus, a tape can not be mounted as the *root*, but can be mounted as some read-only file system. Normally, *root* is disk drive 0, section 0.
- *pipe*-Specifies where pipes are to be allocated (must be a file system-the root file system is normally used).
- *dump*-Specifies the device to be used to dump memory after a system crash. Currently only the TU10 and TU16/TE16 tape drives are supported for this purpose.
- *swap*-Specifies the device and blocks that will be used for *swapping*. *Swplo* is the first block number used and *nswap* indicates how many blocks, starting at *swplo*, to use. Care must be taken that the swap area specified does not overlap any file system. For example, if section 0 is 8,000 blocks long, the *root* file system could occupy the first 6,000 blocks and *swap* the remaining 2,000 by specifying:

  ```
  root rp06 0
  swap rp06 0 6000 2000
  ```

  The VAX release is set up for a *root* of 7,000 blocks and a *swap* of 9,000 blocks.
- *buffers*-Specifies how many *system buffers* to allocate. Real time response improves as more buffers are allocated. UNIX buffers form a "data cache". Improvement in the hit rate of this cache tends to fall as the number of buffers is increased.
- *sabufs*-PDP-11 only: specifies how many *system addressable* buffers to allocate. One buffer is needed for every mounted file system. Certain I/O drivers need such buffers.
- *inodes*-Specifies how many *inode table* entries to allocate. Each entry represents a unique open inode. When the table overflows, the warning message "Inode table overflow" will be printed on the console. The table size should be increased if this happens regularly. The number of entries used depends on the number of active processes, texts, and mounts.
- *files*-Specifies how many *open-file table* entries to allocate. Each entry represents an open file. When the table overflows, the warning message "no file" will be printed on the console. The table size should be increased if this happens regularly.

- *mounts*-Specifies how many *mount table* entries to allocate. Each entry represents a mounted file system. The *root* (/) will always be the first entry. When full, the *mount*(2) system call will return the error EBUSY.
- *coremap*-PDP-11 only: specifies how many entries to allocate to the *list of free memory*. Each entry represents a contiguous group of 64-byte blocks of free memory. When the list overflows, due to excessive fragmentation, the system will undoubtedly crash in an unpredictable manner. The number of entries used depends on the number of processes active, their sizes, and the amount of memory available.
- *swapmap*-Specifies how many entries to allocate to the *list of free swap blocks*. Exactly like the *coremap*, except it represents free blocks in the swap area, in 512-byte units.
- *calls*-Specifies how many *call-out table* entries to allocate. Each entry represents a function to be invoked at a later time by the clock handler. The time unit is 1/60 of a second. The call-out table is used by the terminal handlers to provide terminal delays and by various other I/O handlers. When the table overflows, the system will crash and print the panic message "Timeout table overflow" on the console.
- *procs*-Specifies how many *process table* entries to allocate. Each entry represents an active process. The scheduler is always the first entry and *init*(8) is always the second entry. The number of entries depends on the number of terminal lines available and the number of processes spawned by each user. The average number of processes per user is in the range of 2-5. When full, the *fork*(2) system call will return the error EAGAIN.
- *texts*-Specifies how many *text table* entries to allocate. Each entry represents an active read-only text segment. Such programs are created by using the *-i* or *-n* option of the loader *ld*(1). When the table overflows, the warning message "out of text" is printed on the console.
- *clists*-Specifies how many *character list buffers* to allocate. Each buffer contains up to 24 bytes. The buffers are dynamically linked together to form input and output queues for the terminal lines and various other slow-speed devices. The average number of buffers needed per terminal line is in the range of 5-10. When full, input characters from terminals will be lost and not echoed.
- *maxup*-Specifies how many concurrent processes a user is allowed to run.
- *power*-Specifies whether to attempt restart after a power failure. A value 0 (default) indicates no restart, a value of 1 attempts power-fail restart. On restart, device drivers are called and process 1 (i.e., *init*) is sent a hangup signal; see *init*(8). VAX power-fail is provided for experimental use only in UNIX 3.0.

3.1 UNIX Generation

To generate a new UNIX operating system, make sure that the directories under */usr/src/uts* are up-to-date. Follow the procedure below:

```
cd /usr/src/uts
ed dfile
a
   [information as described above]
.
w
q
make -f uts.mk VER=ver SYS=sys CONFIG=dfile TYPE=type NODE=uucpname
```

Dfile is the complete path name or the path name relative to */usr/src/uts/*/cf* of the file containing the configuration information, *sys* is a system name, *ver* is normally *mmdd* (month and day). The resulting operating system executable file name is the result of concatenating *sys* and *ver* (i.e., "utsa0513"). The *uucpname* is for network identification. *Type* is the CPU type: **i** is used for PDP-11/23 and /34, **vax** is used for VAX, and **id** is used for other CPUs. The procedure will compile *low.s* (*univec.c* on the VAX) and *conf.c*, and load them together with the object libraries into a file called *name*.

The PDP-11 system has a relatively small address space, so that if table sizes or the number of device types are too large, various error messages will result and the above procedure will only create an *a.out* file. In particular, the maximum available data space is 49,152 bytes (57,344 bytes on the PDP-11/23 and the /34). The actual data space requested can be found by using *size*(1) on *a.out* and adding the *data*, *bss*, and, for PDP-11/23 and /34, text segment sizes. One then reduces the specified values for the various system entries until it all fits. The amount of space in the *bss* segment used for each entry is indicated in Section 3.1 above.

When you are satisfied with the new system, you can test it by the following procedure:

```
cd /usr/src/uts
cp sysver /              # sysver as above
cd /
rm /unix
ln /sysver /unix
sync
```

Then halt the processor and reboot the system. Note that this procedure results in two names for the operating system object, the generic */unix*, and the actual name, say */utsa0501*. An old system may be booted by referring to the actual name, but remember that many programs use the generic name */unix* to obtain the *name-list* of the system.

If the new system does not work, verify that the correct device addresses and interrupt vectors have been specified. If the *wrong* interrupt vector and the *correct* device address have been specified for a device, the operating system will print the error message "stray interrupt at XXX" when the device is accessed, where XXX is the correct interrupt vector. If the *wrong* device address is specified, the system will crash with a panic trap of type 0 (indicating a timeout) when the device is accessed.

3.2 Special Files

A special file must be made for every device on your system. Normally, all special files are located in the directory */dev*. Initially, this directory will contain:

console	console terminal
error	see *err*(4)
mem, kmem, null	see *mem*(4)
tty	see *tty*(4)
rp[0-7], rrp[0-7]	disk drive 0, sections 0-7
rl[0-1], rrl[0-1]	disk drives 0 and 1
rk[0-1], rrk[0-1]	disk drives 0 and 1
mt0, rmt0	tape drive 0 (800 bpi)
mt4, rmt4	tape drive 0 (800 bpi, no rewind).

There are two types of special files, block and character. This is indicated by the character *b* or *c* in the listing produced by *ls*(1) with the *-l* flag.

In addition, each special file has a major device number and a minor device number. The major device number refers to the device type and is used as an index into either the *bdevsw* or *cdevsw* table in the configuration file *conf.c*. The minor device number refers to a particular unit of the device type and is used only by the driver for that type. The *config* program with the -t option will list major device numbers.

The program *mknod*(1M) creates special files. For example, the following would create *part* of the initially-supplied */dev* directory:

```
cd /dev
mknod console c 0 0
mknod error c 20 0
mknod mem c 2 0; mknod kmem c 2 1; mknod null c 2 2
mknod tty c 13 0
mknod rp0 b 0 0; mknod rrp0 c 7 0
mknod mt0 b 1 0; mknod rmt0 c 6 0
mknod mt4 b 1 4; mknod rmt4 c 6 4
```

After the special files have been made, their access modes should be changed to appropriate values by *chmod*(1). For example:

```
cd /dev
chmod 622 console
chmod 444 error
chmod 644 mem kmem
chmod 666 null
chmod 666 tty
chmod 400 rp0 rrp0
chmod 666 mt0 rmt0
chmod 666 mt4 rmt4
```

Note that file names have no meaning to the *operating system* itself; only the major and minor device numbers are important. However, many *programs* expect that a particular file is a certain device. Thus, by convention, special files are named as follows:

block device	*conf.c*	*/dev*
RP03 disk	rp	rp*
RP04/5/6 disk	hp	rp*
RS03/4 fixed head disk	hs	rs*
RK05 disk	rk	rk*
RL01 disk	rl	rl*
TU10 tape	tm	mt*
TE/TU16 tape	ht	mt*

character device	*conf.c*	*/dev*
DL11 asynch. line	kl	tty*, console
DH11 asynch. line mux	dh	tty*
DMC11 sync. unit	dmc	dmc*
DZ11 asynch. line mux	dz	tty*
DN11 auto call unit	dn	dn*
DU11 synch. line	du	du*
KMC11 micro	kmc	kmc*
DZ11/KMC11 assist	dza,dzb	tty*
LP11 line printer	lp	lp*
RP03 disk	rp	rrp*
RP04/5/6 disk	hp	rrp*
RS03/4 fixed head disk	hs	rrs*
TU10 tape	tm	rmt*
TE/TU16 tape	ht	rmt*
error	err	error
memory	mm	mem, kmem, null
terminal	sy	tty

For those devices with a */dev* name ending in *, this character is replaced by a string of digits representing the *minor* device number. For example:

```
mknod /dev/mt1 b 1 1
mknod /dev/rp24 b 0 024
```

Note that for disks, an octal number scheme is maintained because each drive is split eight ways. Thus, */dev/rp24* refers to section 4 of physical drive 2. There is also a special file, */dev/swap*, that is used by the program *ps*(1). This file must reflect what device is used for swapping and must be readable. For example:

```
rm /dev/swap
mknod /dev/swap b 0 0
chmod 440 /dev/swap
```

3.3 File Systems

Each physical pack is split into eight logical sections. This split is defined in the operating system by a table with eight entries. Each table entry is two words long. The first specifies how many blocks are in the section, the second specifies the starting cylinder; see *hp*(4) (RP04/5/6) and *rp*(4) (RP03) for default cylinder and block assignments.

These values are described to the system in the header file */usr/include/sys/io.h* which may be changed by using the editor *ed*(1). After such a change, the system must be made again (see Section 3.2 above).

A file system starts at block 0 of a section of the disk and may be as large as the size of that section; if it is smaller than the size of a section, the remainder of that section is unused. Note that the sections themselves may overlap physical areas of the pack, but the file systems must *never* overlap.

The program *mkfs*(1M) is used to initialize a section of the disk to be a file system. Next, the program *labelit*(1M) is used to label the file system with its name and the name of the pack. Finally, the file system may be checked for consistency by using *fsck*(1M). The file system may then be mounted using *mount*(1M).

3.4 DZ11 software with KMC11 assist

KMC microprocessors may be used to control DZ11 asynchronous multiplexers, thus off-loading terminal-oriented functions from the main CPU. The software is distributed in two forms. The KMC11-A version does DMA output of data, character translations, tab expansions, etc. The KMC11-B version does these output functions in addition to doing DMA input of data. Each KMC11 can control up to four DZ11 multiplexers for a total of thirty-two asynchronous lines. Each system can support up to four KMC11 microprocessors. Up to sixty-four DZ11 lines can be controlled by KMC11 microprocessors.

3.4.1 Installation

1. Generate a system (see Section 3.2 above) by including each DZ11 to be controlled by a KMC11 in the configuration file. For example:

   ```
   #   For the KMC11-A
   dza X Y Z

   #   For the KMC11-B
   dzb X Y Z
   ```

   where X is the vector address, Y is the UNIBUS address, and Z is the bus request priority. Also include the KMC11 microprocessors in the configuration file:

   ```
   kmc X Y Z
   ```

2. Install the KMC11 microcode in */lib*:

```
#   For the KMC11-A
cd /usr/src/uts/*/up/dza/dzkload
/lib/cpp dza.s | kas -o /lib/dzkmc.o

#   For the KMC11-B
cd /usr/src/uts/*/up/dzb/dzkload
/lib/cpp main.s | kasb -o /lib/dzkmc.o
```

3. Copy *dzkload* into */etc*:

```
#   For the KMC11-A
cp /usr/src/uts/*/up/dza/dzkload /etc

#   For the KMC11-B
cp /usr/src/uts/*/up/dzb/dzkload /etc
```

4. Update */etc/rc* to execute *dzkload* for multi-user and power-fail *init* states. Each KMC11 used to control a DZ11 must be loaded with microcode. For each KMC11 used to control a DZ11 include:

```
/etc/dzkload /dev/kmc?
```

   where **?** is the minor device number of that KMC11.

5. Special files (see Section 3.3 above) must be created for each KMC11 and DZ11 line:

```
#   Example
/etc/mknod /dev/kmc? c X ?
/etc/mknod /dev/tty?? c Y Z
```

   X is the major device number of the KMC11 and **?** is the minor device number of the KMC11 controlling the DZ11 multiplexers, i.e., the KMC11 loaded with microcode in step 4. Y is the major device number of the *dza/dzb* device as is supplied by *config*(1M). Z is the minor device number for a particular line on a DZ11. This number is composed of three fields. The low-order three bits are the line number relative to a DZ11. The next three bits contain the minor device number of the DZ11 controlling these lines. Note that this number is the absolute DZ11 number, not the number relative to the KMC11. The high-order two bits are the minor device number of the KMC11 controlling this DZ11. For example:

```
mknod /dev/tty?? c Y 0241
```

   specifies the second line (0 through 7) on the fifth DZ11 to be controlled by the KMC11 with minor device number 2. The DZ11 number is specified by the order of appearance in the configuration file. The first four DZ11 multiplexers must be associated with one KMC11 and the next four must be associated with another KMC11. The order in which the KMC11 microprocessors are specified is not significant.

4. ADMINISTRATIVE FILES

4.1 /etc/motd

This file contains the *message-of-the-day*. It is printed by */etc/profile* after every successful *login*.

4.2 /etc/rc

On the transition between init states, */etc/init* invokes */bin/sh* to run */etc/rc* (must have executable modes). So that */etc/rc* can properly handle the removal of temporary files and the mounting and unmounting of file systems, it is invoked with three arguments: new state, number of times this state has been entered, previous state. When the system is initially booted, */etc/rc* is invoked with arguments "1 0 1"; when state two(multi-user) is subsequently entered, the

arguments are "2 0 1".

Daemons may be invoked either by */etc/rc* or by including lines for them in */etc/inittab*.

The */etc/rc* file is also used to initialize KMC11 microprocessors (see */etc/dzkload* and*/etc/vpmload* below).

This file must be edited to suit local conditions; see *init*(8).

4.3 /etc/inittab

This file indicates to */etc/init* which processes to create in each init state. By convention, state 1 is *single-user* and state 2 is *multi-user*. For example, the following line creates the single-user environment:

```
1:co:c:env HOME=/ PATH=/bin:/etc:/usr/bin /bin/sh </dev/console\
        >/dev/console 2>/dev/console
1:xx:k:/etc/getty console ! 0
```

This indicates that for state 1 a process with the arbitrary unique identifier *co* should be created. The program invoked for this process should be the shell and when it exits it should be reinvoked (*c* flag).

To attach a *login* process to the console in the multi-user state, add the line:

```
2:co:c:/etc/getty console 4
```

and for line */dev/tty00* for use by 300/150/110 baud terminals, add the following line:

```
2:00:c:/etc/getty tty00 0 60
```

The arguments to *getty* are the device, speed table, and number of seconds to allow before hanging up the line.

Again, this file must be edited for local conditions; see *getty*(8), *init*(8), and *inittab*(5).

4.4 /etc/dzkload

This file is invoked as a command by */etc/rc*. It contains instructions for initializing a KMC11 microprocessor which is to function as a controller for one or more DZ11 communications multiplexers (see Section 3.5 above). This file must be edited to suit the configuration.

4.5 /etc/passwd

This file is used to describe each *user* to the system. You must add a new line for each new user. Each line has seven fields separated by colons:

1. Login name: normally 1-6 characters, first character alphabetic, the remainder alphanumeric, no upper-case characters.
2. Encrypted password: initially null, filled in by *passwd*(1). The encrypted password contains 13 bytes, while the actual password is limited to a maximum of 8 bytes. The encrypted password may be followed by a comma and up to 4 more bytes of password "age" information.
3. User ID: a number between 0 and 65,535; 0 indicates the super-user. These other IDs are reserved:

bin::2:	software administration;
sys::3:	system operation;
adm::4:	system administration;
uucp::5:	UNIX-to-UNIX file copy;
rje::68:	remote job entry administration;
games::196:	miscellaneous; never a real user.

4. Group ID: the default is group 1 (one).

5. Accounting information: this field is used by various accounting programs. It usually contains the user name, department number, and account number.
6. Login directory: full path-name (keep them reasonably short).
7. Program name: if null, */bin/sh* is invoked after a successful login. If present, the named program is invoked in place of */bin/sh*.

For example:

```
ghh::38:1:6824-G.H.Hurtz(4357):/usr/ghh:
grk::44:1:6510-S.P.LeName(4466):/usr/grk:/bin/rsh
```

See also *passwd*(5), *login*(1), *passwd*(1).

4.6 /etc/group

This file is used to describe each *group* to the system. You must add a new line for each new group. Each line has four fields separated by colons:

1. Group name: normally 1-6 characters, first character alphabetic, rest alphanumeric, no upper-case characters.
2. Encrypted password: initially null, filled in by *passwd*(1). The encrypted password contains 13 bytes, while the actual password is limited to a maximum of 8 bytes.
3. Group ID: a number between 0 and 65,535.
4. Login names: list of all *login* names in the group, separated by commas.

We strongly discourage group passwords. See also *group*(5).

4.7 /etc/profile

When the shell is executed and is the leader of a process group, as is the case when it is invoked by *login*, it will read and execute the commands in */etc/profile* before executing commands in the user's *.profile* file. This allows the system administrator to set up a standard environment for all users (e.g., executing *umask*, setting shell variables, etc.) and take care of other housekeeping details (such as *news -n*).

4.8 /etc/checklist

This file contains a list of default devices to be checked for consistency by the *fsck*(1M) program. The devices normally correspond to those mounted when the system is in *multi-user* mode. For example, a sample checklist would be:

```
/dev/rp0
/dev/rrp1
```

4.9 /etc/shutdown

This file contains procedures to gracefully shut down the system in preparation for file-save or scheduled down-time.

4.10 /etc/filesave.?

This file contains the detailed procedures for the local file-save.

4.11 /usr/adm/pacct

This file contains the process accounting information; see *acct*(1M).

4.12 /usr/adm/wtmp

This file is the log of login processes.

5. REGENERATING SYSTEM SOFTWARE

System source is issued under the directory */usr/src*. The sub-directories are named *cmd* (commands), *lib* (libraries), *uts* (the operating system), *head* (header files), and *util* (utilities);

see *mk*(8) for details on how to remake system software.

A couple of anomalies: the accounting routine that deals with holidays and the prime/non-prime time split must be recompiled at the end of each year (it is currently correct for BTL-Murray Hill in 1980). The file is */usr/src/cmd/acct/lib/pnpsplit.c*. A reminder is sent to */usr/adm/acct/nite/log*, the standard place for such messages, starting a week before year-end and continuing until *pnpsplit.c* is recompiled.

5.1 PDP-11 Command Regeneration

The distributed object code has been compiled for machines without separate "I/D" space and without floating-point hardware. If your system has separate I/D space (i.e., is a PDP-11/70 or PDP-11/45), you should *mkcmd* adb, awk, bs, cc, cpio, dc, du, dump, efl, f77, fgrep, find, fsck, lex, make, mkfs, nm, pcat, restor, spell, spline, tbl, tplot, troff, units, unpack, uucp, volcopy, and yacc. If your configuration has an FP11-[ABC] floating-point processor (or the compatible 11/23 chip), you should *mkcmd* acct, adb, awk, bs, cc, pack, spline, tplot, typo, and units. If your configuration has both separate I/D space and floating point, you should *mkcmd* acct, adb, awk, bs, cc, cpio, dc, du, dump, efl, f77, fgrep, find, fsck, lex, make, mkfs, nm, pack, pcat, restor, spell, spline, tbl, tplot, troff, typo, units, unpack, uucp, volcopy, and yacc.

6. FILE SYSTEM CONVERSION TO UNIX (VAX)

Procedures have been developed for converting UNIX file systems from PDP-11/UNIX (including UNIX/TS, PWB Edition 2.0, and Research Version 7) to VAX UNIX. Direct conversion from other systems (i.e., Version 6-based or UNIX/RT) is also possible, but the administrator should get help.

The following UNIX commands are referenced in this section: *cpio*(1), *find*(1), *fsck*(1M), *fscv*(1M), *mkdir*(1), *mkfs*(1M), *mount*(1M), and *umount*(1M). The reader is assumed to be familiar with them.

Unless you have repealed Murphy's Law, you should allow plenty of time for the conversion. As a lower bound, it takes about two hours to convert each 65K of file system space.

6.1 Preliminaries

Obviously, the new system should be generated and decently tested before conversion is attempted. Source for local commands and libraries should be moved to the VAX and compiled and tested. Your users may also reasonably require time to develop conversion programs for data files that contain binary information.

6.1.1 The Old System

The file systems should be pruned of marginal files. The following shell sequence will get rid of all executables:

```
#   For each user file system:
find /usr-fs-list -type f -print | xargs file | \
      sed -n -e 's/\([^:]*\):.*executable/\1/p' >/usr/tmp/exfiles
#   You may want to look this file over before the next step.
xargs rm -f</usr/tmp/exfiles
rm /usr/tmp/exfiles
```

6.1.2 Spare Packs

Do not convert without spare packs-that is courting disaster. It is best to keep the old packs for several days, and to make backup tapes as well.

6.2 Converting the Hard Way

Using *find* and *cpio* takes much longer, but you will have optimized converted user file systems when you are finished (compacted *inodes* and directories, file and free-list blocks arranged for fastest access).

6.2.1 Copying from the Old System

The following steps should be executed by the super-user on an idle, stand-alone (old) system:

```
#   For each user file system:
cd /file-system-name
find . -cpio /dev/rmt1
```

The *-cpio* option of *find* produces ten-block records on physical tape in *cpio* format. Unless there are a great many linked files, a 1,600-bpi, 2,400-foot reel should hold about 65K file system blocks. If you have larger file systems, the easiest (fastest, safest) thing to do is to use a raw disk pack in place of the tape (i.e., **/dev/rrp??** in place of the **/dev/rmt1** above). In our tests, multi-reel *find/cpio* tapes have worked. *Find* can also be used to pick up *parts* of file systems that can be combined later as described below.

6.2.2 Under the New System

Re-create each file system as follows:

```
mkdir /file-system-name
mkfs /dev/rrp? blocks:inodes 7 418
#   The magic numbers "7 418" above refer to free-list ordering:
#   (rotational angle of 7 and 418 blocks/cylinder for RP04/5/6s)
labelit /dev/rrp? file-system-name pack-num
mount /dev/rp? /file-system-name
cd /file-system-name
#   Mount tape created during step 3
cpio -idmB </dev/rmt1
#   If you are combining the smaller file systems,
#   you may copy-in more than one tape per new file system
#   (but make sure that first-level directory names are unique)
```

After the tapes have been copied in, the new file systems should be unmounted and checked (using *fsck*(1M)).

6.3 Converting the Easy Way

The *fscv*(1M) command has been provided for fast conversion between PDP-11 and VAX file systems.

Note that *fscv* will not convert "special files" (user file systems only). *Fscv* source will compile and run on either system. It was designed primarily for use in computer labs where there are a mixture of PDP-11 and VAX systems.

Example 1:

```
#   Converting PDP-11 to VAX:
#   Make sure that file system cylinder boundaries agree!
fscv -v /dev/rrp21 /dev/rrp31
```

Example 2:

```
#   Converting "in place" to the PDP-11:
#   Anyone who does this without making a copy first deserves
#   whatever bad (plenty) that can happen!
fscv -p /dev/rrp12 /dev/rrp12
```

See the *fscv*(1M) manual entry.

6.4 A Final Precaution

It is only sensible to do another complete file system backup under the new operating system (using another set of tapes or packs).

Administrative Advice for UNIX

R. C. Haight

Bell Laboratories
Murray Hill, New Jersey 07974

The material presented here is based on the author's experiences and opinions. Nevertheless, it may prove useful. The material on phototypesetting was contributed by D. W. Smith.

1. ADMINISTRATOR'S ROAD MAP

Getting started as a UNIX† system administrator is hard work. There are no real shortcuts to a working knowledge of the system. You will need time for reading, study and hands-on experimenting. Don't commit yourself to "going live" with your system until you have had two weeks to teach yourself your job, and get the initial hardware quirks ironed-out.

Don't consign the *Setting Up UNIX* document to oblivion after your initial system "gen". In addition to needing it again whenever you add/change equipment, you will find that it contains valuable material about system tuning (buffers, *clist*s, etc.) that appears nowhere else.

As an administrator, you should be familiar with a lot of the distributed documentation. The *Internals, Operations, and Administration* papers from *Documents for UNIX* should all be studied, as well as the *Introduction*, *How to Get Started*, and most of the entries of the *UNIX User's Manual*. In that manual, you should pay special attention to: *acct**(1M), *chmod*(1), *chown*(1), *config*(1M), *cpio*(1), *date*(1), *df*(1), *du*(1), *ed*(1), *env*(1), *find*(1), *fsck*(1M), *kill*(1), *mail*(1), *mkdir*(1), *mkfs*(1M), *ncheck*(1M), *ps*(1), *rm*(1), *rmdir*(1), *shutdown*(1M), *stty*(1), *su*(1), *sync*(1M), *time*(1), *volcopy*(1M), *wall*(1M), *who*(1), and *write*(1) in Section 1; all of Section 4; *acct*(5) in Section 5; and *crash*(8) and *vaxops*(8) in Section 8.

2. SYSTEM CAPACITY

The figures below are approximations based on our experience over several years:

Hardware Configuration	*Number of Simultaneous Users*
PDP-11/23; 256K-byte memory; 2 RL01 disks*	4
PDP-11/34; 256K-byte memory with cache; 2 RL01 disks*	8
PDP-11/45; 248K-byte memory; RP03 disk*	16
Above with RP06 (RP04, RP05) disk*	20
Above with memory cache	25
PDP-11/70; 512K-byte memory; RP06 (RP04, RP05) disks* (2 or more drives)	32
Above with 768K-byte memory and a disk drive (or fixed-head disk) set aside for the root file system	40
VAX-11/780; 1M-byte memory; at least 3 RP06 disks*	48

* Or equivalent.

† UNIX is a Trademark of Bell Laboratories.

See *Setting Up UNIX* for the list of supported hardware options.

3. DISK FREE SPACE

Making files is easy under UNIX. It has been said that the only standard thing about all UNIX systems is the message-of-the-day telling users to clean up their files. Administratively, both free disk blocks and free inodes (UNIX talk for file headers) can be a problem. If the free inode count falls below 100, the system spends most of its time rebuilding the free inode array. If a file system runs out of space, the system prints "no-space" messages and does little else. To avoid problems, the following start-of-day free counts should be maintained:

- The file system containing */tmp* (temporary files):
  — 16-user system: 1,500 free blocks.
  — 40-user system: 3,000 free blocks.
- The file system containing */usr*:
  — 3,000 to 6,000 free blocks, depending on load.
- Other user file systems:
  — 6% to 10% free, depending on user habits (3,000 blocks minimum).

This brings up an associated problem: how big should file systems be? Our preference is to set aside space on each drive for a copy of root/swap and use the rest of the pack for a single file system. However, if you have user groups that fight over disk space, it may be better to split them up arbitrarily (i.e., divide a pack into more than one file system). Warning: if you set up different disk drives with differing cylinder partitions between file systems, it will probably lead to an operations goof someday.

4. A VERY FEW WORDS ABOUT SYSTEM TUNING

- As shipped, UNIX has *no* programs with the text-bit mode set (see *chmod*(1)). The top contenders for the *t*-bit are *nroff* and *troff* followed (generally) by the larger phases of the C compiler (including the assembler and loader). The *t*-bit is only meaningful with pure text programs (*ld*(1) options *-i* or *-n*). Don't overdo it, and keep *t*-bit programs in the root file system.
- File system reorganization (described below) can help throughput, but at the expense of down time. If you do it when your users are all asleep, it can help.
- If you use normal *shutdown* and *filesave.u* procedures, the file system check program (*fsck*(1M), *-S* option) will help keep the disk free list in reasonable order.
- Try to keep disk drive usage balanced. If you have over 20 users, the *root* file system (*/bin*, */tmp*, */etc*, and *swap*) deserves a drive of its own.
- If you have a noisy modem (poorly executed do-it-yourself null-modem) or a disconnected modem cable, UNIX will spend a lot of CPU time trying to get it logged in. A random check of systems uncovers a lot of this going on.

5. WHY YOU MUST HAVE A SPARE DISK DRIVE

- Without a spare disk drive, the system will be *down* when a drive is *down*.
- Without a spare drive, it is difficult to reorganize file systems or to restore user files.

6. DISK PACKS

- Buy only fully ECC correctable packs and test them.
- If a pack develops uncorrectable errors, recondition it, or get rid of it.

RP06 disk packs used with UNIX need not be totally error-free, but must be "flag-free". The term flag-free means that there should be no unrecoverable ECC (Error Correcting Code) errors. Technically, proper ECC handling can recover from 11-bit error bursts. However, we hear that the length of bursts can grow as a pack ages. We recommend that no pack that has more than 8-bit error bursts be accepted. For the PDP-11, the following explanation may help

(paraphrased from a DEC source).

In reading the formatter printout, ECC correctable errors are identified by the headings "DATA ERROR DURING WRITE CHECK." Error-register values are printed below the message. The two registers of interest are RPER1 and RPEC2. A RPER1 value of 1000000 indicates ECC (no other bits on). The RPEC2 register describes the bit span of the error. For example, RPEC2=003774 means that there was an unacceptable 9-bit (binary 0000011111111100) error burst; RPEC2=000240 is an acceptable 3-bit span (0000000010100000-there may be zero bits mixed in). If such acceptable errors account for all "unrecoverable" errors reported (and there aren't too many of them), then you have a flag-free pack.

On the VAX, even this scant information was not available, so we have written our own formatter (it tells its tale in English); see *rp6fmt*(8). We plan to make this program available in the future (along with other UNIX-oriented diagnostics) for the PDP-11 as well.

7. PROTECTING USER FILES

Users, especially inexperienced ones, occasionally remove their own files. Open files are sometimes lost when the system crashes. Once in a great while, an entire file system will be destroyed (picture a disk controller that goes bad and writes when it should read). Here is a suggested file backup procedure:

- Each day, copy all user file systems to backup packs. Keep these packs 3 to 5 days before re-using them.
- Once a week, copy each file system to tape. Keep weekly tapes for 8 weeks.
- Keep bi-monthly tapes "forever" (they should be re-copied once a year).

The most recent weekly tapes should be kept off premises. The other tapes should be in a fire-proof safe, if you can afford one.

When UNIX goes down, active files can get scrambled. Your users will not want to start the day over every time your system fails. In addition to good backup, you *must* have file-system patching expertise available (on-site or on-call). If you ever re-boot the system for general use without checking out the file systems, terrible things will happen (we once had five duplicate entries on a file-system free list-this ruined over 100 new files in just three days). Study *fsck*(1M) and *crash*(8), as well as *FSCK-The UNIX/TS File System Check Program*.

8. UNIX FILE SYSTEM BACKUP PROGRAMS

The following backup programs are distributed:

- *Dump/restor*: This is a familiar tape-based system that has been used for several years. Full dumps are usually taken when the *dump* program warns that an incremental dump will run to more than one reel.
- *Find/cpio*: UNIX is distributed in *cpio* format. The *-cpio* option of the *find* command has made it time-competitive with *dump/restor*. However, it does not produce a "perfect" restore from a full dump plus incremental dump (new and changed files are OK, but file removal information is lost). Because of this, full dumps should be taken fairly often (weekly/bi-weekly). *Cpio* is the only program listed here that makes system-independent copies. It can be used to move files between various versions of UNIX/RT and UNIX, and can be used in system conversion.
- *Volcopy*: physical file system copying to disk or tape. For those who can afford a spare drive, *volcopy* to disk provides convenient file restore and quick recovery from disk disasters (remember the spare drive). Tape *volcopy* provides good long-term backup because the file system can be read-in fairly quickly, mounted, and browsed over. Disk and tape *volcopy* are generally used together for short- and long-term backup. *Volcopy* can also be used for full dumps with either *dump/restor* or *cpio/find*.

The table below summarizes attributes of these programs. The file system size is 65,500 blocks in all cases; times are in minutes; judgements are subjective.

	dump/restor	*find/cpio*	*volcopy (disk)*	*volcopy (tape)*
Full dump time	40	40	2	15
Incremental dump time	6	7	-	-
Full restore time	40(?)	80	2	15
Incremental restore time	8	10	-	-
Ease of restoring:				
one file	fair	fair	good	fair
a directory	poor	fair	good	good
scattered files	poor	poor	good	good
full restore	fair	fair	very good	good
Needs tape drive	yes	yes	no	yes
Needs spare file system (only when restoring)	no	no	-	yes
Needs spare disk drive (two CPUs can share)	-	-	yes	-
Maintains pack/tape labels	no	no	yes	yes
Handles multi-reel tape	yes	yes	-	yes
512 blocks per record	1,10	1,10	88	10
Interactive (i.e., ties up console)	no(?)	yes	yes	yes
May require separate I/D space	no	no	no*	no

* Blocks per record are cut to 22 without separate I/D space.

We strongly recommend the spare disk drive: as explained in Section 5 above, the speed and convenience of *volcopy* are by no means the only advantage of a spare drive.

9. CONTROLLING DISK USAGE

If your UNIX system is a success, you will soon run out of disk space:

- During the considerable delay before you can get more drives, you will need to control usage:
  - — Try to maintain the start-of-day counts recommended above. Watch usage during the day by executing the *df* command regularly.
  - — The *du*(1) command should be executed (after hours) regularly (e.g., daily) and the output kept (in an accessible file) for later comparison. In this way you can spot users who are rapidly increasing their disk usage.
  - — The *find*(1) can be used to locate inactive (or large) files. Example:

    ```
    find / -mtime +90 -atime +90 -print >somefile
    ```

    records in "somefile" the names of files neither written nor accessed in the last 90 days. Of course, this works best if you are super-user.
- You will also have to balance usage between file systems. To do this you will have to move user directories. Users should be taught to accept file system name changes (and to program around them-preferably ahead of time). The user's login directory name (available in the shell variable *HOME*) should be utilized to minimize path name dependencies. User groups with more extensive file system structures should set up a shell variable to refer to the file system name (e.g.: *FS*).
- The *find*(1) and *cpio*(1) commands can be used to move user directories and to manipulate the file system tree. The following sequence is useful (it moves, via magnetic tape, the directory trees *userx* and *usery* from file system *filesys1* to file system *filesys2* where, presumably, more space is available):

```
cd /filesys1
find userx usery -cpio /dev/rmt0
cd /filesys2
mkdir userx usery
chown userx userx
chown usery usery
cpio -idmB </dev/rmt0
#  Make sure new copy is OK
#  Change userx and usery login directories in the /etc/passwd file
rm -rf /filesys1/userx /filesys1/usery
```

When moving more than one user in this way:
— Keep users with common interests in the same file system (they may have linked files).
— Move groups of users who may have linked files with a single *cpio* (otherwise linked files will be unlinked and duplicated).

10. REORGANIZING FILE SYSTEMS

The procedure for moving users described above can be expanded to provide a way to reorganize whole file systems. Reorganization can improve system response time. This is particularly true of the *root* file system (which must be reorganized with all other file systems unmounted) and */usr*. Unfortunately, reorganization of large file systems is slow.

11. KEEPING DIRECTORY FILES SMALL

Directories larger than 5K bytes (320 entries) are very inefficient because of file system indirection. A UNIX user once complained that it took the system ten minutes to complete the login process; it turned out that his login directory was 25K bytes long, and the login program spent that time fruitlessly looking for a non-existent *.profile* file. A large */usr/mail* or */usr/spool/uucp* directory can also really slow the system down. The following will ferret out such directories:

```
find / -type d -size +10 -print
```

Removing files from directories does not make the directories get smaller (the empty directory entries are available for reuse). The following will "compact" */usr/mail* (or any other directory):

```
mv /usr/mail /usr/omail
mkdir /usr/mail
chmod 777 /usr/mail
cd /usr/omail
find . -print | cpio -plm ../mail
cd ..
rm -rf omail
```

12. ADMINISTRATIVE USE OF "CRON"

The program *cron*(1M) is useful in the administration of the system; it can be used to:

- Turn off the programs in directory */usr/games* during prime time.
- Run programs off-hours:
  — accounting;
  — file system administration;
  — long-running, user-written shell procedures (using the *su*(1) command), for example:

```
su - userx userx_shell arg ...
```

13. WATCH OUT FOR FILES AND DIRECTORIES THAT GROW

- Accounting files:
  — */usr/adm/wtmp*-login information;
  — */usr/adm/pacct*-process accounting; gets big quickly.
- Other files:
  — */usr/lib/cronlog*-status log of commands executed by *cron*(1M);
  — */usr/spool*-spooling directory for line printers, *uucp*(1C), etc., and whose sub-directories should be compacted as described above.

14. ALLOCATING RESOURCES TO USERS

A prospective user should obtain connect-time and file-space authorization through appropriate channels. Once this is done, the user should apply for a login by providing the following information to the "system administrator":

- User's name.
- Suggested login name (not more than 8 characters, beginning with a lower-case letter).
- Relationships to other users (this influences the choice of the file system).
- Estimate of required file space (this also influences the choice of the file system).

Users should be forced to have passwords (not more than 8 characters long, but more than 5, and *not* in Webster's Unabridged); *passwd*(5) explains how to do that.

15. THE MATTER OF ACCOUNTING AND USAGE

You should run the accounting programs even if you do not "bill" for service. Otherwise, your users' habits (especially *bad* habits) will be a mystery to you. Accounting information can also help you find performance bottlenecks, unused logins, bad phone lines, etc.

16. DIAL LINE UTILIZATION

If prime-time dial line utilization gets much over 70%, users will start to encounter busy signals when dialing in. This, in turn, will lead to "line hogging". The only solutions are to get a larger (another) machine, or to get rid of users. Manual policing will help some, but "automatic" policing will be *invariably* subverted by users.

17. "BIRD-DOGGING"

When the system is busy (lines busy and/or slow response), someone should determine why this is so. The *who*(1) command lists the people logged in. The *ps*(1) command shows what they are doing. (The */etc/whodo* command combines the output of *who* and *ps*.) Unfortunately, *ps* operates from heuristics that can consistently fail to report certain processes in a busy system. That is, one must be careful about hanging up an apparently inactive line. The *acctcom*(1M) command can read the shell accounting file */usr/adm/pacct* backwards from the most recent entry. It will print entries for selected lines or login names.

18. 300/1,200-BAUD TERMINALS

Don't use upper-case-only terminals. Get full-duplex, full-ASCII terminals. Hardware horizontal tabbing is very desirable, because it increases output speed and lowers system overhead. A fair proportion of your terminals should provide for correspondence-quality hard-copy output to take advantage of the UNIX word-processing capabilities; see *term*(7).

19. LINE PRINTERS

Most line printers are troublesome and impose considerable overhead on the system. Most also lack hardware tabs, character overstrike capability, etc. A printer that will work over an asynchronous link (DC1/DC3 protocol required) may be the best bet.

20. SECURITY

The current UNIX is not tamper-proof. You can't keep people from "breaking" the system, but you can usually detect that they have done so. The following command will mail (to root) a list of all "set user ID" programs owned by *root* (super-user):

```
find / -user root -perm -4100 -exec ls -l {} \; | mail root
```

Any surprises in *root*'s mail should be investigated. Related advice:

- Change the super-user password regularly. Don't pick obvious passwords (choose 6-to-8 character nonsense strings that combine alphabetics with digits or special characters).
- If you have dial ports and do not *require* passwords, you are courting trouble.
- The *chroot*(1M) ans *su*(1) commands are inherently dangerous, as are *group* passwords; consider removing them from "production" systems.
- Login directories, *.profile* files, and files in */bin*, */usr/bin*, */lbin*, and */etc* that are writable by others than their respective owners are security weak spots; police your system regularly against them.
- Remember, no time-sharing system with dial ports is really secure. Don't keep top-secret stuff on the system.

21. COMMUNICATING WITH YOUR USERS

The directory */usr/news* and the *news*(1) command are provided as a way to get *brief* announcements to your users. More pressing items (one-liners) can be entered in the */etc/motd* (message of the day) file; *motd* and (new to the user) *news* are announced at login time.

To reach users who are already logged in, use the *wall*(1M) (write all) command. Don't use *wall* while logged-in as super-user, except in emergencies.

The */usr/news* directory should be cleaned out every few weeks so that nothing older than, say, three months is ever found there. The *motd* file should be cleaned out daily.

We have found that, on most systems, a file in */usr/news* will reach 50% of the users within a day and over 80% of the users within a week.

22. TROUBLESHOOTING

It would be easy to write a book on this topic. The following are some of the key items:

a. Dealing with the hardware service contractor:

- Before you take out a hardware service contract (with DEC or with someone else), be sure that the contractor agrees to get along with the UNIX software ("It's the hardware," says you; "It's the software," says the hardware service contractor).
- Keep on top of problems. For instance, DEC has a problem-aging priority scheme. Find out about any such scheme that your contractor may have, and make them prove that it is being followed. Remember that an unreported problem is getting no priority at all. If a problem persists, escalate it up your contractor's local management chain; it may also be effective to complain to your contractor's sales representative.
- If you are serious about service to your users, you should have an extended-period service contract (e.g., 16 hours/day, 6 days/week). Arrange for preventive maintenance, non-critical repair, and add-on installation work to be done before or after prime time.
- If you have a service contract, learn the details. In particular, make sure that preventive maintenance is scheduled in advance and that it is completed.
- Ask the hardware service contractor to provide and maintain a "site log". You will have to work on the log, as well.
- Make sure that your hardware vendor (as well as your hardware service contractor, if

the two are different) agrees to the presence of non-DEC equipment on your system (even if you have none to start with).

- Run error logging. Keep console sheets. Make sure error messages are shown to your contractor's Customer Engineers.
- Take core dumps after system crashes and interpret results for Customer Engineers.
- Keep down-time records and make sure that your hardware service contractor knows about them.

b. Dealing with the telephone services vendor:

You are most apt to have telephone problems when you rearrange or add equipment. You may also have occasional central office, trunking, and modem failures:

- Be specific with repair operators: tell them that the trouble involves *data* equipment.
- If your first call fails to get results, ask for the "supervisor" on the second call, and, if necessary, escalate further to get the problem solved.

c. Some obvious problem areas:

- Disk Drives-Over 50% of your problems are likely to be related to the disk subsystem. As mentioned earlier, the way to keep your system *up* is to have a spare disk drive. Remember:
  — Preventive maintenance of disk drives is very important.
  — Make sure that the Customer Engineers who service your hardware see the error-logging printouts and console error messages produced by UNIX (and that they understand them).
  — Disk failure can ruin a UNIX file system. The *only* defense is to make a complete, daily file backup! (See *Protecting User Files* above.)
  — Many administrators believe the the RP04 disk drives fail more often than RP06s and take longer to fix.
- Dial Ports-In this area, as well as in the area of synchronous data interfaces, there is room for finger-pointing among all your vendors. Check for obvious things:
  — Is the system in "multi-user" mode?
  — Is the */etc/inittab* file OK?
  — Are any cables loose (*both* ends)?
  — In some telephone offices, trunk-hunting is based on 10-number groups. Hunting *between* such groups can fail independently of anything else.

  The possibilities for trouble are many. The "decision table" below attempts to describe some alternatives; it is meant primarily for users of DH11/DZ11 asynchronous devices. If you are unfamiliar with the format, (vertical) Rule 3 reads: "If line rings *and* ring light shows *and* computer does *not* answer *and* switching the modem solves the problem, then it is likely to be a telephone company problem; also, busy out that line."
- Early experience with the DZ11 has been poor. Several different problems have cropped up including bad line units and a stuck interrupt bit that crashes the system. Don't install DZs without giving them the full diagnostic treatment.
- Synchronous Ports-High-speed synchronous interface devices are even more trouble than dial equipment. The following is a list of potential trouble spots:
  — Your UNIX software.
  — Your interface device (e.g., DQS11B).
  — Cable to your modem.
  — Your modem.
  — The communications line.
  — Other modem.
  — Other cable.
  — Other interface device.
  — Other system's software.

  Think of the finger-pointing possibilities. The best defense is a good line monitor.

- Power Supply Modules-There are a lot of them, and they do fail, more or less regularly. Hard failure can be detected at the console; voltage drift is tougher. Failure of the FP11 (floating-point unit) power supply can be slow to fix, because Customer Engineers are likely to work back from the far end of the "bus", taking a long time to find the problem.

Asynchronous Line Problems

Rules:	1	2	3	4	5	6	7	8	9	0
Condition:										
Line rings	N	Y	Y	Y	Y	Y	Y	Y	Y	Y
Ring light shows on telephone console	-	N	Y	Y	Y	Y	Y	Y	Y	Y
Computer answers	-	-	N	N	Y	Y	Y	Y	Y	Y
Login message received on terminal	-	-	-	-	N	N	Y	Y	Y	Y
Switching modem solves problem	-	-	Y	N	Y	N	-	-	-	-
User can login	-	-	-	-	-	-	N	N	N	Y
Telephone console shows data received	-	-	-	-	-	-	Y	Y	N	-
Problem affects whole DH/DZ (up to 16 lines)	-	-	-	-	-	-	Y	N	-	-
Diagnosis and/or Action:										
No problem	-	-	-	-	-	-	-	-	-	X
PDP-11 hardware problem likely	-	-	-	X	-	X	X	-	-	-
Telephone problem likely	X	X	X	-	X	-	-	-	X	-
May be a problem with user's terminal	-	-	-	-	-	-	-	X	-	-
Busy out bad line(s)	X	X	X	X	X	X	X	-	X	-

23. DATASET OPTIONS

The following dataset options seem to work with UNIX:

The 801C-L1 (Auto-Call Unit):

Jumpers:

E2 to *E3*
E6 to *E5*

Options:

Y, X, T, B,
ZG, ZP, G,
R, ZT

Switches (0 = open, 1= closed, i.e., side next to number is down):

S1 = 1000[1] (Bracketed switches are missing on some models.)
S2 = 0101
S3 = 11010
S4 = 11[00]

The 212A-L1 (1,200-baud full-duplex):

Options:

E, ZF, YF, YC,
YG, YJ, YK,
S, V, A, T, ZH,
W, YP, YR

Switches:

S1 = [0]001
S2 = 110001000
S3 = 11110000
S5 = 00

24. NULL MODEM WIRING

Improperly wired null modems can cause spurious interrupts, especially at higher baud rates. A single bad modem on a 9,600-baud line can waste 15% of your CPU power. The following (symmetrical) wiring plan will prevent such problems:

pin 1 to 1
pin 2 to 3
pin 3 to 2
strap pin 4 to 5 in the same plug
pin 6 to 20
pin 7 to 7
pin 8 to 20
pin 20 to 6 and 8
ground unused pins

25. 113D, 103J DATA SET PROBLEMS

The DH11 and DJ11 multiplexers normally have a jumper connecting pin 25 to pin 4 (request to send), thus asserting pin 25 when the line is opened. This jumper should be removed for any lines connected to 113Ds or 103Js (also applies to 103Js with 801s).

26. PHOTOTYPESETTING EQUIPMENT AND SUPPLIES

Read this section if you plan to use the phototypesetting software of UNIX.

Phototypesetter. The phototypesetter and fonts currently supported by UNIX are manufactured by:

Wang Graphic Systems, Inc.
Executive Drive
Hudson, NH 03051 (603-889-8550)

The phototypesetter is an on-line C/A/T System 1 with a high-speed turret. The external paper tape reader on the typesetter is not needed, because the typesetter is connected to the PDP-11 CPU via a DR11C.

PDP 11/45 Only. The following modification (developed by DEC Field Service) should be made to the DR11C (without this modification, the system may crash when the typesetter is powered down): "Add two 390-ohm resistors-from E-18 pin 6 to ground, and from E-18 pin 3 to ground. Put a piece of insulating tubing over the leads so that they do not short out the 'etch' runs that they cross."

Fonts. There are eight fonts that are normally used, as shown in the table below. The first three of the these provide the most-often used (serif) typeface. The last three are used when a sans-serif typeface is desired. The fourth font contains a number of Greek characters and mathematical symbols; see *NROFF/TROFF User's Manual* by J. F. Ossanna. The fifth font is useful for typesetting text that you wish to look like terminal or printer output, e.g., for examples of programs. Wang Graphic Systems, Inc. offers a variety of other fonts. For *troff* to be able to use these fonts, corresponding font tables must be built and compiled into the directory */usr/lib/font*.

Name	*Part Number*	*Troff Name*
BT Times Roman	802-016A	R
Times Italic	802-013A	I
Times Bold	802-014A	B
BT PI Font #4 Special Characters	829-021B	S
BT PI Font #6 Constant Width	829-046A	CW
Geneva (Helvetica) Regular	803-032B	G or H
Geneva (Helvetica) Regular Italic	803-033B	GI or HI
Geneva (Helvetica) Medium	803-034B	GM or HM

Other fonts for which the source font tables are supplied are:

Name	*Troff Name*
Boston Condensed	BC
News Condensed	C
Century Schoolbook Expanded	CE
Century Schoolbook Italic	CI
Century Bold Italic	CK
Century Schoolbook	CS
Futura (Utica) Demibold	FD or UD
Text Greek	GR
Geneva Light	L
Geneva Light Italic	LI
Palatino	PA
Palatino Bold	PB
Palatino Italic	PI
Stymie Bold	SB
Stymie Medium Italic	SI
Stymie Medium	SM

Paper and Chemicals. The phototypesetter "prints" onto photo-mechanical paper, which can be obtained from a photographic supply house and is specified as:

- Kodak Ektamatic Paper, Grade S, Type 2250, 8 in. × 150 ft., Spec. 175 (or equivalent).

Also obtainable from such a supply house are the chemicals for the developing process:

- Kodak Ektamatic A10 Activator (or equivalent).
- Kodak Ektamatic S40 Stabilizer (or equivalent).

These chemicals should be ordered in 9.5-liter (2.5-gallon) containers for the circulator.

Developer. A Kodak Ektamatic Processor Model 214K (or equivalent) is used to process the paper from the typesetter. A light-proof box attached to the 214K (to hold the output cassette from the typesetter) is called an "Autofeeder" and can be obtained from:

Peripheral Graphics, Inc.
Andover Industrial Center
York Street
Andover, MA 01810 (617-475-9005)

Circulator and Dryer. A circulator and a paper dryer, as well as a shelf for the dryer can be obtained from:

Mohr Enterprises
8015 North Ridgeway Ave.
Skokie, IL 60076 (312-674-8890)

The needed parts are:

- ME-8 Mohrflow: circulator to increase the usability of the chemicals.
- ME-5 Mohrdry: dryer for the photo-mechanical paper.
- Dryer Extension: shelf to support the dryer; it connects to the circulator cabinet.

Also obtainable from Mohr Enterprises are cleaners for the developer and circulator. Such cleaning is needed every 2 to 4 weeks, depending on the volume of work:

- R-53 Mohrchem Activator Cleaner Concentrate.
- R-57 Mohrchem Stabilizer Cleaner Concentrate.

Each quart bottle makes 9.5 liters (2.5 gallons) of reusable cleaner to clean the tubing, rollers, and tray of the developer and circulator. Equivalent cleaners can also be obtained elsewhere.

June 1980

The PWB/UNIX Accounting System

Henry S. McCreary

Bell Laboratories
Piscataway, New Jersey 08854

ABSTRACT

The PWB/UNIX* Accounting System provides methods to collect per-process resource utilization data, record connect sessions, monitor disk utilization, and charge fees to specific logins. A set of C 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.

---

* UNIX is a Trademark of AT&T Bell Laboratories.

The PWB/UNIX Accounting System

Henry S. McCreary

Bell Laboratories
Piscataway, New Jersey 08854

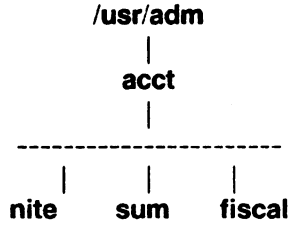
1. Introduction

The PWB/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 Kernal 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 (UID 4) is used by the accounting system and has the following directory structure:



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*.

---

* 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 output.
3. For a complete explanation of the files used by the accounting system, see Attachment 1.

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 cleanup 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 is instructed by */etc/rc* to 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:
   /usr/lib/acct/shutacct
3. Three entries should be made in **/usr/lib/crontab** so that *cron* will automatically start some shell procedures.
   ```
   0 4 * * 1-6 /bin/su - adm -c "/usr/lib/acct/runacct 2> /usr/adm/acct/nite/fd2log"
   0 2 * * 4 /bin/su - adm -c "/usr/lib/acct/sdisk"
   5 * * * * /bin/su - adm -c "/usr/lib/acct/ckpacct"
   ```
4. The **PATH** shell variable in **adm**'s **.profile** should be set 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 in the following order:

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* which are connect accounting records.[5]

---

4. Files used by *runacct* are assumed to be in the **nite** directory unless otherwise noted.
5. Accounting records are in *tacct.h* format.

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.

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.
Examples:

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 filesave backup. However, certain files must be fixed in order to maintain the integrity of the accounting system.

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.

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 userids, or a userid of 65535. First check **sum/tacctprev** with *prtacct*. If it looks ok, 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. Recompiling pnpsplit

The *pnpsplit* subroutine is used by *acctcon1* and *acctprc1* to determine the difference between prime and non-prime time. Prime time is defined as 9 a.m. to 5 p.m. Monday through Friday, holidays excluded. Every year on the day after Christmas, the following message will appear in **log**:

***** RECOMPILE pnpsplit WITH NEW HOLIDAYS *****

Unfortunately, this will cause *runacct* to fail 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** array to reflect the new holidays. **Pnpsplit.c** is in **/usr/src/cmd/acct/lib**.
2. Recompile the accounting library *a.a*, *acctprc1*, and *acctcon1* by:

   super-user to root
   cd /usr/src
   ARGS="library acctcon1 acctprc1" ./:mkcmd acct

10. Summary

The PWB/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 pages 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.

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*
log*MMDD*	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
wtmperror*MMDD*	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 *?*

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
 */

/*
 *      following taken from (or modified versions of) <sys/types.h>
 */
typedef unsigned short  dev_t;
typedef unsigned int    ino_t;
typedef long            off_t;
typedef long            time_t;

/*
 *      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.
#define KCORE(clicks)   ((double) clicks/16)
#define SECSINDAY       86400L
```

Format of pacct files (acct.h)

```
/*
 * Accounting structures
 */

typedef unsigned short comp_t; /* "floating point" */

struct  acct
{
        char    ac_flag;        /* Accounting flag */
        char    ac_stat;        /* Exit status */
        short   ac_uid;                 /* Accounting user ID */
        short   ac_gid;                 /* Accounting group ID */
        dev_t   ac_tty;                 /* control typewriter */
        time_t  ac_btime;               /* Beginning time */
        comp_t ac_utime;                /* Accounting user time */
        comp_t ac_stime;                /* Accounting system time */
        comp_t ac_etime;                /* Accounting elapsed time */
        comp_t ac_mem;                          /* memory usage */
        comp_t ac_io;
        comp_t ac_rw;
        char    ac_comm[8];             /* command name */
};

extern  struct  acct    acctbuf;
extern  struct  inode   *acctp;         /* inode of accounting file */

#defineAFORK 01                 /* has executed fork, but no exec */
#defineASU   02                 /* used super-user privileges */
#defineACCTF 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 */
};
```

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	
00 00 00	pvg	79 06 07	whr	79 06 08	systst
00 00 00	rakesh	79 06 07	wwe	79 06 08	teach
00 00 00	rfg	79 06 08	?	79	
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				

January 1980

FSCK-The UNIX/TS File System Check Program

T. J. Kowalski

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

When a UNIX/TS 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.

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]

---

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/TS or a *sync* command is issued.

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.

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.

---

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

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.

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 of information (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/TS 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.

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 to the file system 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/TS.

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., and Olsson, S. B. eds., *UNIX/TS User's Manual, Edition 1.1* (January 1978).

[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 in 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, and -t. *Fsck* terminates on this error condition. See the *fsck*(1M) manual entry for further detail.

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 detail.

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 detail.

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.

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.

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 Section 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.

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 Section 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 Section 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.

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 Section 4.4 and 4.2.2.

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 detail.

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 detail.

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 Section 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.

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 Section 4.2.2 and 4.4.

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 Section 4.2.2 and 4.4.

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 Section 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 Section 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.

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 Section 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 Section 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 Section 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 Section 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 Section 4.1.2, 4.2.3, and 4.2.4.

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 detail.

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/TS is not rebooted immediately, the work done by *fsck* may be undone by the in-core copies of tables UNIX/TS 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/TS rebooted. If UNIX/TS is not rebooted immediately, the work done by *fsck* may be undone by the in-core copies of tables UNIX/TS keeps.

May 1979

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 I/O System

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

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

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

Device Classes

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

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

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

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

Overview of I/O

The purpose of the *open* and *creat* system calls is to set up entries in three separate system tables. The first of these is the *u_ofile* table, which is stored in the system's per-process data area *u*. This table is indexed by the file descriptor returned by the *open* or *creat*, and is accessed during a *read*, *write*, or other operation on the open file. An entry contains only a pointer to the corresponding entry of the *file* table, which is a per-system data base. There is one entry in the *file* table for each instance of *open* or *creat*. This table is per-system because the same instance of an open file must be shared among the several processes which can result from *fork*s after

†UNIX is a Trademark of Bell Laboratories.

the file is opened. A *file* table entry contains flags which indicate whether the file was open for reading or writing or is a pipe, and a count which is used to decide when all processes using the entry have terminated or closed the file (so the entry can be abandoned). There is also a 32-bit file offset which is used to indicate where in the file the next read or write will take place. Finally, there is a pointer to the entry for the file in the *inode* table, which contains a copy of the file's i-node.

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

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

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

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

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

Character Device Drivers

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

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

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

flag which is non-zero if the file was open for writing in the process which performs the final *close.*

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

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

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

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

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

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

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

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

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

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

characters.

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

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

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

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

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

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

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

The Block-device Interface

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

residual word count used to communicate information from an I/O routine to its caller. Finally, there is a flag word with bits indicating the status of the buffer. These flags will be discussed below.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Block Device Drivers

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

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

Although the most usual argument to the strategy routines is a genuine buffer header allocated as discussed above, all that is actually required is that the argument be a pointer to a place containing the appropriate information. For example the *swap* routine, which manages movement of core images to and from the swapping device, uses the strategy routine for this

device. Care has to be taken that no extraneous bits get turned on in the flag word.

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

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

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

Raw Block-device I/O

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

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

UNIX 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.

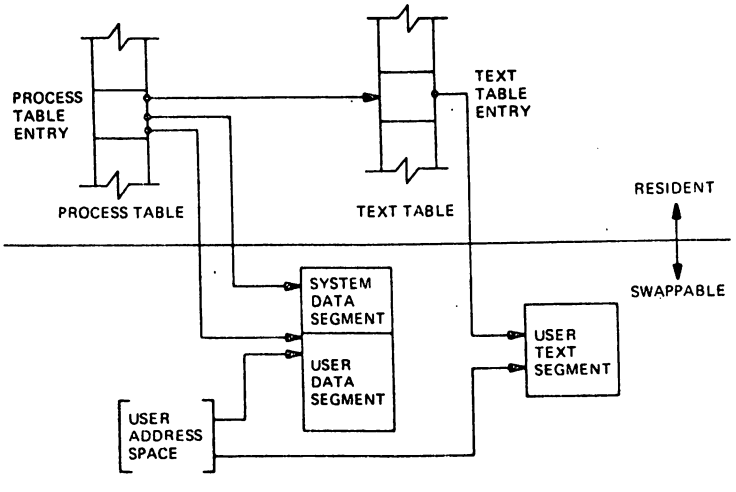


Fig. 1—Process control data structure.

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 **execs** the second program. This is analogous to a "goto." If a program wishes to regain control after **exec**ing a second program, it should **fork** a child process, have the child **exec** the second program, and have the parent **wait** for the child. This is analogous to a "call." Breaking up the call into a binding followed by a transfer is similar to the subroutine linkage in SL-5.[1]

2.2. Swapping

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

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

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

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

2.3. Synchronization and scheduling

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

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

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

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

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

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

3. I/O SYSTEM

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

I/O" is a complete misnomer.

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

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

3.1. Block I/O system

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

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

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

3.2. Character I/O system

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

3.2.1. Disk drivers

Disk drivers are implemented with a queue of transaction records. Each record holds a read/write flag, a primary memory address, a secondary memory address, and a transfer byte count. Swapping is accomplished by passing such a record to the swapping device driver. The block I/O interface is implemented by passing such records with requests to fill and empty system buffers. The character I/O interface to the disk drivers create a transaction record that points directly into the user area. The routine that creates this record also insures that the user is not swapped during this I/O transaction. Thus by implementing the general disk driver, it is possible to use the disk as a block device, a character device, and a swap device. The only really disk-specific code in normal disk drivers is the pre-sort of transactions to minimize latency for a particular device, and the actual issuing of the I/O request.

3.2.2. Character lists

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

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

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

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

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

3.2.3. Other character devices

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

4. THE FILE SYSTEM

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

The UNIX file system is a disk data structure accessed completely through the block I/O system. As stated before, the canonical view of a "disk" is a randomly addressable array of 512-byte blocks. A file system breaks the disk into four self-identifying regions. The first block (address 0) is unused by the file system. It is left aside for booting procedures. The second block (address 1) contains the so-called "super-block." This block, among other things, contains the size of the disk and the boundaries of the other regions. Next comes the i-list, a list of file definitions. Each file definition is a 64-byte structure, called an i-node. The offset of a particular i-node within the i-list is called its i-number. The combination of device name (major and minor numbers) and i-number serves to uniquely name a particular file. After the i-list, and to the end of the disk, come free storage blocks that are available for the contents of files.

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

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

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

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

4.1. File system implementation

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

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

The user process accesses the file system with certain primitives. The most common of these are **open**, **create**, **read**, **write**, **seek**, and **close**. The data structures maintained are shown in Fig. 2.

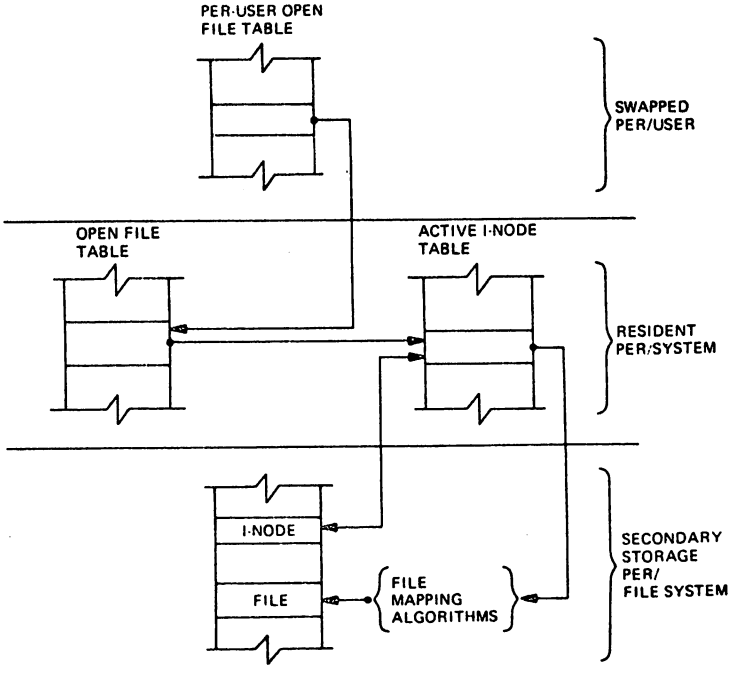


Fig. 2—File system data structure.

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

With file sharing, it is necessary to allow related processes to share a common I/O pointer

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

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

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

4.2. Mounted file systems.

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

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

4.3. Other system functions

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

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

code. The idea of implementing "system" code with general user primitives comes directly from MULTICS.[6]

References

1. R. E. Griswold and D. R. Hanson, "An Overview of SL5," *SIGPLAN Notices* **12**(4) pp. 40-50 (April 1977).
2. E. W. Dijkstra, "Cooperating Sequential Processes," pp. 43-112 in *Programming Languages*, ed. F. Genuys, Academic Press, New York (1968).
3. J. A. Hawley and W. B. Meyer, "MUNIX, A Multiprocessing Version of UNIX," M.S. Thesis, Naval Postgraduate School, Monterey, Cal. (1975).
4. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Bell Sys. Tech. J.* **57**(6) pp. 1905-1929 (1978).
5. S. R. Bourne, "UNIX Time-Sharing System: The UNIX Shell," *Bell Sys. Tech. J.* **57**(6) pp. 1971-1990 (1978).
6. E. I. Organick, *The MULTICS System*, M.I.T. Press, Cambridge, Mass. (1972).

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 requres hard intellectual effort to convert, since the algorithms embodied in this part of the code are typically complicated and machine dependent.

To summarize, however, if you need a C compiler written for a machine with a reasonable architecture, the compiler is already three quarters finished!

Overview

This paper discusses the structure and organization of the portable compiler. The intent is to give the big picture, rather than discussing the details of a particular machine implementation. After a brief overview and a discussion of the source file structure, the paper describes the major data structures, and then delves more closely into the two passes. Some of the theoretical work on which the compiler is based, and its application to the compiler, is discussed elsewhere.[4] One of the major design issues in any C compiler, the design of the calling sequence and stack frame, is the subject of a separate memorandum.[5]

The compiler consists of two passes, *pass1* and *pass2*, that together turn C source code into assembler code for the target machine. The two passes are preceded by a preprocessor, that handles the **#define** and **#include** statements, and related features (e.g., **#ifdef**, etc.). It is a nearly machine independent program, and will not be further discussed here.

The output of the preprocessor is a text file that is read as the standard input of the first pass. This produces as standard output another text file that becomes the standard input of the second pass. The second pass produces, as standard output, the desired assembler language source code. The preprocessor and the two passes all write error messages on the standard error file. Thus the compiler itself makes few demands on the I/O library support, aiding in the bootstrapping process.

Although the compiler is divided into two passes, this represents historical accident more than deep necessity. In fact, the compiler can optionally be loaded so that both passes operate in the same program. This "one pass" operation eliminates the overhead of reading and writing the intermediate file, so the compiler operates about 30% faster in this mode. It also occupies about 30% more space than the larger of the two component passes.

Because the compiler is fundamentally structured as two passes, even when loaded as one, this document primarily describes the two pass version.

The first pass does the lexical analysis, parsing, and symbol table maintenance. It also constructs parse trees for expressions, and keeps track of the types of the nodes in these trees. Additional code is devoted to initialization. Machine dependent portions of the first pass serve to generate subroutine prologs and epilogs, code for switches, and code for branches, label definitions, alignment operations, changes of location counter, etc.

The intermediate file is a text file organized into lines. Lines beginning with a right parenthesis are copied by the second pass directly to its output file, with the parenthesis stripped off. Thus, when the first pass produces assembly code, such as subroutine prologs, etc., each line is prefaced with a right parenthesis; the second pass passes these lines to through to the assembler.

The major job done by the second pass is generation of code for expressions. The expression parse trees produced in the first pass are written onto the intermediate file in Polish Prefix form: first, there is a line beginning with a period, followed by the source file line number and name on which the expression appeared (for debugging purposes). The successive lines represent the nodes of the parse tree, one node per line. Each line contains the node number, type, and any values (e.g., values of constants) that may appear in the node. Lines representing nodes with descendants are immediately followed by the left subtree of descendants, then the right. Since the number of descendants of any node is completely determined by the node

number, there is no need to mark the end of the tree.

There are only two other line types in the intermediate file. Lines beginning with a left square bracket ('[') represent the beginning of blocks (delimited by { ... } in the C source); lines beginning with right square brackets (']') represent the end of blocks. The remainder of these lines tell how much stack space, and how many register variables, are currently in use.

Thus, the second pass reads the intermediate files, copies the ')' lines, makes note of the information in the '[' and ']' lines, and devotes most of its effort to the '.' lines and their associated expression trees, turning them turns into assembly code to evaluate the expressions.

In the one pass version of the compiler, the expression trees that are built by the first pass have been declared to have room for the second pass information as well. Instead of writing the trees onto an intermediate file, each tree is transformed in place into an acceptable form for the code generator. The code generator then writes the result of compiling this tree onto the standard output. Instead of '[' and ']' lines in the intermediate file, the information is passed directly to the second pass routines. Assembly code produced by the first pass is simply written out, without the need for ')' at the head of each line.

The Source Files

The compiler source consists of 22 source files. Two files, *manifest* and *macdefs*, are header files included with all other files. *Manifest* has declarations for the node numbers, types, storage classes, and other global data definitions. *Macdefs* has machine-dependent definitions, such as the size and alignment of the various data representations. Two machine-independent header files, *mfile1* and *mfile2*, contain the data structure and manifest definitions for the first and second passes, respectively. In the second pass, a machine dependent header file, *mac2defs*, contains declarations of register names, etc.

There is a file, *common*, containing (machine independent) routines used in both passes. These include routines for allocating and freeing trees, walking over trees, printing debugging information, and printing error messages. There are two dummy files, *comm1.c* and *comm2.c*, that simply include *common* within the scope of the appropriate pass1 or pass2 header files. When the compiler is loaded as a single pass, *common* only needs to be included once: *comm2.c* is not needed.

Entire sections of this document are devoted to the detailed structure of the passes. For the moment, we just give a brief description of the files. The first pass is obtained by compiling and loading *scan.c*, *cgram.c*, *xdefs.c*, *pftn.c*, *trees.c*, *optim.c*, *local.c*, *code.c*, and *comm1.c*. *Scan.c* is the lexical analyzer, which is used by *cgram.c*, the result of applying *Yacc*[6] to the input grammar *cgram.y*. *Xdefs.c* is a short file of external definitions. *Pftn.c* maintains the symbol table, and does initialization. *Trees.c* builds the expression trees, and computes the node types. *Optim.c* does some machine independent optimizations on the expression trees. *Comm1.c* includes *common*, that contains service routines common to the two passes of the compiler. All the above files are machine independent. The files *local.c* and *code.c* contain machine dependent code for generating subroutine prologs, switch code, and the like.

The second pass is produced by compiling and loading *reader.c*, *allo.c*, *match.c*, *comm1.c*, *order.c*, *local.c*, and *table.c*. *Reader.c* reads the intermediate file, and controls the major logic of the code generation. *Allo.c* keeps track of busy and free registers. *Match.c* controls the matching of code templates to subtrees of the expression tree to be compiled. *Comm2.c* includes the file *common*, as in the first pass. The above files are machine independent. *Order.c* controls the machine dependent details of the code generation strategy. *Local2.c* has many small machine dependent routines, and tables of opcodes, register types, etc. *Table.c* has the code template tables, which are also clearly machine dependent.

Data Structure Considerations.

This section discusses the node numbers, type words, and expression trees, used throughout both passes of the compiler.

The file *manifest* defines those symbols used throughout both passes. The intent is to use the same symbol name (e.g., MINUS) for the given operator throughout the lexical analysis, parsing, tree building, and code generation phases; this requires some synchronization with the *Yacc* input file, *cgram.y*, as well.

A token like MINUS may be seen in the lexical analyzer before it is known whether it is a unary or binary operator; clearly, it is necessary to know this by the time the parse tree is constructed. Thus, an operator (really a macro) called UNARY is provided, so that MINUS and UNARY MINUS are both distinct node numbers. Similarly, many binary operators exist in an assignment form (for example, $-=$), and the operator ASG may be applied to such node names to generate new ones, e.g. ASG MINUS.

It is frequently desirable to know if a node represents a leaf (no descendants), a unary operator (one descendant) or a binary operator (two descendants). The macro *optype(o)* returns one of the manifest constants LTYPE, UTYPE, or BITYPE, respectively, depending on the node number *o*. Similarly, *asgop(o)* returns true if *o* is an assignment operator number ($=$, $+=$, etc.), and *logop(o)* returns true if *o* is a relational or logical (&&, ||, or !) operator.

C has a rich typing structure, with a potentially infinite number of types. To begin with, there are the basic types: CHAR, SHORT, INT, LONG, the unsigned versions known as UCHAR, USHORT, UNSIGNED, ULONG, and FLOAT, DOUBLE, and finally STRTY (a structure), UNIONTY, and ENUMTY. Then, there are three operators that can be applied to types to make others: if *t* is a type, we may potentially have types *pointer to t*, *function returning t*, and *array of t's* generated from *t*. Thus, an arbitrary type in C consists of a basic type, and zero or more of these operators.

In the compiler, a type is represented by an unsigned integer; the rightmost four bits hold the basic type, and the remaining bits are divided into two-bit fields, containing 0 (no operator), or one of the three operators described above. The modifiers are read right to left in the word, starting with the two-bit field adjacent to the basic type, until a field with 0 in it is reached. The macros PTR, FTN, and ARY represent the *pointer to*, *function returning*, and *array of* operators. The macro values are shifted so that they align with the first two-bit field; thus PTR+INT represents the type for an integer pointer, and

$$\text{ARY} + (\text{PTR}<<2) + (\text{FTN}<<4) + \text{DOUBLE}$$

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 functionis allocated, and the function return value is copied into it on return; the function returns the address of that region. This is simple to implement, but is 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. When this is done, there is a match: in fact,

reg += *b*

simply describes the effect of the add instruction on a typical machine. After the add is emitted, the tree is rewritten to consist merely of the register node, since the result of the add is now in the register. This agrees with the cookie passed to the second invocation of *order*, so this invocation terminates, returning to the first level. The original tree has now become

a = **reg**

which matches a template for the store instruction. The store is output, and the tree rewritten to become just a single register node. At this point, since the top level call to *order* was interested only in side effects, the call to *order* returns, and the code generation is completed; we have generated a load, add, and store, as might have been expected.

The effect of machine architecture on this is considerable. For example, on the Honeywell 6000, the machine dependent heuristics recognize that there is an "add to storage" instruction, so the strategy is quite different; *b* is loaded in to a register, and then an add to storage instruction generated to add this register in to *a*. The transformations, involving as they do the semantics of C, are largely machine independent. The decisions as to when to use them, however, are almost totally machine dependent.

Having given a broad outline of the code generation process, we shall next consider the

heart of it: the templates. This leads naturally into discussions of template matching and register allocation, and finally a discussion of the machine dependent interfaces and strategies.

The Templates

The templates describe the effect of the target machine instructions on the model of computation around which the compiler is organized. In effect, each template has five logical sections, and represents an assertion of the form:

> **If** we have a subtree of a given shape (1), and we have a goal (cookie) or goals to achieve (2), and we have sufficient free resources (3), **then** we may emit an instruction or instructions (4), and rewrite the subtree in a particular manner (5), and the rewritten tree will achieve the desired goals.

These five sections will be discussed in more detail later. First, we give an example of a template:

```
ASG PLUS,   INAREG,
            SAREG,      TINT,
            SNAME,      TINT,
                        0,          RLEFT,
                        "           add         AL,AR\n",
```

The top line specifies the operator (+=) and the cookie (compute the value of the subtree into an AREG). The second and third lines specify the left and right descendants, respectively, of the += operator. The left descendant must be a REG node, representing an A register, and have integer type, while the right side must be a NAME node, and also have integer type. The fourth line contains the resource requirements (no scratch registers or temporaries needed), and the rewriting rule (replace the subtree by the left descendant). Finally, the quoted string on the last line represents the output to the assembler: lower case letters, tabs, spaces, etc. are copied *verbatim* to the output; upper case letters trigger various macro-like expansions. Thus, **AL** would expand into the 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 += 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 += 1) − 1)

which preserves the semantics. Once again, this is not too attractive for the most common cases, but can generate close to optimal code when the type of x is unusual.

Finally, the indirection (UNARY MUL) operator is also handled in a special way. The machine dependent routine *offstar* is extremely important for the efficient generation of code. *Offstar* is called with a tree that is the direct descendant of a UNARY MUL node; its job is to transform this tree so that the combination of UNARY MUL with the transformed tree becomes addressable. On most machines, *offstar* can simply compute the tree into an A or B register, depending on the architecture, and then *canon* will make the resulting tree into an OREG. On many machines, *offstar* can profitably choose to do less work than computing its entire argument into a register. For example, if the target machine supports OREGS with a constant offset from a register, and *offstar* is called with a tree of the form

expr + const

where *const* is a constant, then *offstar* need only compute *expr* into the appropriate form of register. On machines that support double indexing, *offstar* may have even more choice as to how to proceed. The proper tuning of *offstar*, which is not typically too difficult, should be one of the first tries at optimization attempted by the compiler writer.

The Sethi-Ullman Computation

The heart of the heuristics is the computation of the Sethi-Ullman numbers. This computation is closely linked with the rewriting rules and the templates. As mentioned before, the Sethi-Ullman numbers are expected to estimate the number of scratch registers needed to compute the subtrees without using any stores. However, the original theory does not apply to real machines. For one thing, the theory assumes that all registers are interchangeable. Real machines have general purpose, floating point, and index registers, register pairs, etc. The

theory also does not account for side effects; this rules out various forms of pathology that arise from assignment and assignment ops. Condition codes are also undreamed of. Finally, the influence of types, conversions, and the various addressability restrictions and extensions of real machines are also ignored.

Nevertheless, for a "useless" theory, the basic insight of Sethi and Ullman is amazingly useful in a real compiler. The notion that one should attempt to estimate the resource needs of trees before starting the code generation provides a natural means of splitting the code generation problem, and provides a bit of redundancy and self checking in the compiler. Moreover, if writing the Sethi-Ullman routines is hard, describing, writing, and debugging the alternative (routines that attempt to free up registers by stores into temporaries "on the fly") is even worse. Nevertheless, it should be clearly understood that these routines exist in a realm where there is no "right" way to write them; it is an art, the realm of heuristics, and, consequently, a major source of bugs in the compiler. Often, the early, crude versions of these routines give little trouble; only after the compiler is actually working and the code quality is being improved do serious problem have to be faced. Having a simple, regular machine architecture is worth quite a lot at this time.

The major problems arise from asymmetries in the registers: register pairs, having different kinds of registers, and the related problem of needing more than one register (frequently a pair) to store certain data types (such as longs or doubles). There appears to be no general way of treating this problem; solutions have to be fudged for each machine where the problem arises. On the Honeywell 66, for example, there are only two general purpose registers, so a need for a pair is the same as the need for two registers. On the IBM 370, the register pair (0,1) is used to do multiplications and divisions; registers 0 and 1 are not generally considered part of the scratch registers, and so do not require allocation explicitly. On the Interdata 8/32, after much consideration, the decision was made not to try to deal with the register pair issue; operations such as multiplication and division that required pairs were simply assumed to take all of the scratch registers. Several weeks of effort had failed to produce an algorithm that seemed to have much chance of running successfully without inordinate debugging effort. The difficulty of this issue should not be minimized; it represents one of the main intellectual efforts in porting the compiler. Nevertheless, this problem has been fudged with a degree of success on nearly a dozen machines, so the compiler writer should not abandon hope.

The Sethi-Ullman computations interact with the rest of the compiler in a number of rather subtle ways. As already discussed, the *store* routine uses the Sethi-Ullman numbers to decide which subtrees are too difficult to compute in registers, and must be stored. There are also subtle interactions between the rewriting routines and the Sethi-Ullman numbers. Suppose we have a tree such as

$$A - B$$

where A and B are expressions; suppose further that B takes two registers, and A one. It is possible to compute the full expression in two registers by first computing B, and then, using the scratch register used by B, but not containing the answer, compute A. The subtraction can then be done, computing the expression. (Note that this assumes a number of things, not the least of which are register-to-register subtraction operators and symmetric registers.) If the machine dependent routine *setbin*, however, is not prepared to recognize this case and compute the more difficult side of the expression first, the Sethi-Ullman number must be set to three. Thus, the Sethi-Ullman number for a tree should represent the code that the machine dependent routines are actually willing to generate.

The interaction can go the other way. If we take an expression such as

$$*(p + i)$$

where p is a pointer and i an integer, this can probably be done in one register on most machines. Thus, its Sethi-Ullman number would probably be set to one. If double indexing is possible in the machine, a possible way of computing the expression is to load both p and i into

registers, and then use double indexing. This would use two scratch registers; in such a case, it is possible that the scratch registers might be unobtainable, or might make some other part of the computation run out of registers. The usual solution is to cause *offstar* to ignore opportunities for double indexing that would tie up more scratch registers than the Sethi-Ullman number had reserved.

In summary, the Sethi-Ullman computation represents much of the craftsmanship and artistry in any application of the portable compiler. It is also a frequent source of bugs. Algorithms are available that will produce nearly optimal code for specialized machines, but unfortunately most existing machines are far removed from these ideals. The best way of proceeding in practice is to start with a compiler for a similar machine to the target, and proceed very carefully.

Register Allocation

After the Sethi-Ullman numbers are computed, *order* calls a routine, *rallo*, that does register allocation, if appropriate. This routine does relatively little, in general; this is especially true if the target machine is fairly regular. There are a few cases where it is assumed that the result of a computation takes place in a particular register; switch and function return are the two major places. The expression tree has a field, *rall*, that may be filled with a register number; this is taken to be a preferred register, and the first temporary register allocated by a template match will be this preferred one, if it is free. If not, no particular action is taken; this is just a heuristic. If no register preference is present, the field contains NOPREF. In some cases, the result must be placed in a given register, no matter what. The register number is placed in *rall*, and the mask MUSTDO is logically or'ed in with it. In this case, if the subtree is requested in a register, and comes back in a register other than the demanded one, it is moved by calling the routine *rmove*. If the target register for this move is busy, it is a compiler error.

Note that this mechanism is the only one that will ever cause a register-to-register move between scratch registers (unless such a move is buried in the depths of some template). This simplifies debugging. In some cases, there is a rather strange interaction between the register allocation and the Sethi-Ullman number; if there is an operator or situation requiring a particular register, the allocator and the Sethi-Ullman computation must conspire to ensure that the target register is not being used by some intermediate result of some far-removed computation. This is most easily done by making the special operation take all of the free registers, preventing any other partially-computed results from cluttering up the works.

Compiler Bugs

The portable compiler has an excellent record of generating correct code. The requirement for reasonable cooperation between the register allocation, Sethi-Ullman computation, rewriting rules, and templates builds quite a bit of redundancy into the compiling process. The effect of this is that, in a surprisingly short time, the compiler will start generating correct code for those programs that it can compile. The hard part of the job then becomes finding and eliminating those situations where the compiler refuses to compile a program because it knows it cannot do it right. For example, a template may simply be missing; this may either give a compiler error of the form "no match for op ..." , or cause the compiler to go into an infinite loop applying various rewriting rules. The compiler has a variable, *nrecur*, that is set to 0 at the beginning of an expressions, and incremented at key spots in the compilation process; if this parameter gets too large, the compiler decides that it is in a loop, and aborts. Loops are also characteristic of botches in the machine-dependent rewriting rules. Bad Sethi-Ullman computations usually cause the scratch registers to run out; this often means that the Sethi-Ullman number was underestimated, so *store* did not store something it should have; alternatively, it can mean that the rewriting rules were not smart enough to find the sequence that *sucomp* assumed would be used.

The best approach when a compiler error is detected involves several stages. First, try to get a small example program that steps on the bug. Second, turn on various debugging flags in

the code generator, and follow the tree through the process of being matched and rewritten. Some flags of interest are −e, which prints the expression tree, −r, which gives information about the allocation of registers, −a, which gives information about the performance of *rallo*, and −o, which gives information about the behavior of *order*. This technique should allow most bugs to be found relatively quickly.

Unfortunately, finding the bug is usually not enough; it must also be fixed! The difficulty arises because a fix to the particular bug of interest tends to break other code that already works. Regression tests, tests that compare the performance of a new compiler against the performance of an older one, are very valuable in preventing major catastrophes.

Summary and Conclusion

The portable compiler has been a useful tool for providing C capability on a large number of diverse machines, and for testing a number of theoretical constructs in a practical setting. It has many blemishes, both in style and functionality. It has been applied to many more machines than first anticipated, of a much wider range than originally dreamed of. Its use has also spread much faster than expected, leaving parts of the compiler still somewhat raw in shape.

On the theoretical side, there is some hope that the skeleton of the *sucomp* routine could be generated for many machines directly from the templates; this would give a considerable boost to the portability and correctness of the compiler, but might affect tunability and code quality. There is also room for more optimization, both within *optim* and in the form of a portable "peephole" optimizer.

On the practical, development side, the compiler could probably be sped up and made smaller without doing too much violence to its basic structure. Parts of the compiler deserve to be rewritten; the initialization code, register allocation, and parser are prime candidates. It might be that doing some or all of the parsing with a recursive descent parser might save enough space and time to be worthwhile; it would certainly ease the problem of moving the compiler to an environment where *Yacc* is not already present.

Finally, I would like to thank the many people who have sympathetically, and even enthusiastically, helped me grapple with what has been a frustrating program to write, test, and install. D. M. Ritchie and E. N. Pinson provided needed early encouragement and philosophical guidance; M. E. Lesk, R. Muha, T. G. Peterson, G. Riddle, L. Rosler, R. W. Mitze, B. R. Rowland, S. I. Feldman, and T. B. London have all contributed ideas, gripes, and all, at one time or another, climbed "into the pits" with me to help debug. Without their help this effort would have not been possible; with it, it was often kind of fun.

References

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey (1978).
2. S. C. Johnson, "Lint, a C Program Checker," Comp. Sci. Tech. Rep. No. 65 (1978).
3. A. Snyder, *A Portable Compiler for the Language C,* Master's Thesis, M.I.T., Cambridge, Mass. (1974).
4. S. C. Johnson, "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages,* pp. 97-104 (January 1978).
5. M. E. Lesk, S. C. Johnson, and D. M. Ritchie, *The C Language Calling Sequence,* Bell Laboratories internal memorandum (1977).
6. S. C. Johnson, "Yacc — Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey (July 1975).
7. A. V. Aho and S. C. Johnson, "Optimal Code Generation for Expression Trees," *J. Assoc. Comp. Mach.* **23**(3) pp. 488-501 (1975). Also in *Proc. ACM Symp. on Theory of Computing,* pp. 207-217, 1975.
8. R. Sethi and J. D. Ullman, "The Generation of Optimal Code for Arithmetic Expressions," *J. Assoc. Comp. Mach.* **17**(4) pp. 715-728 (October 1970). Reprinted as pp. 229-247 in *Compiler Techniques,* ed. B. W. Pollack, Auerbach, Princeton NJ (1972).
9. A. V. Aho, S. C. Johnson, and J. D. Ullman, "Code Generation for Machines with Multiregister Operations," *Proc. 4th ACM Symp. on Principles of Programming Languages,* pp. 21-28 (January 1977).

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.

April 3, 1978

Password Security: A Case History

Robert Morris

Ken Thompson

Bell Laboratories
Murray Hill, New Jersey 07974

INTRODUCTION

Password security on the UNIX† time-sharing system [1] is provided by a collection of programs whose elaborate and strange design is the outgrowth of many years of experience with earlier versions. To help develop a secure system, we have had a continuing competition to devise new ways to attack the security of the system (the bad guy) and, at the same time, to devise new techniques to resist the new attacks (the good guy). This competition has been in the same vein as the competition of long standing between manufacturers of armor plate and those of armor-piercing shells. For this reason, the description that follows will trace the history of the password system rather than simply presenting the program in its current state. In this way, the reasons for the design will be made clearer, as the design cannot be understood without also understanding the potential attacks.

An underlying goal has been to provide password security at minimal inconvenience to the users of the system. For example, those who want to run a completely open system without passwords, or to have passwords only at the option of the individual users, are able to do so, while those who require all of their users to have passwords gain a high degree of security against penetration of the system by unauthorized users.

The password system must be able not only to prevent any access to the system by unauthorized users (i.e. prevent them from logging in at all), but it must also prevent users who are already logged in from doing things that they are not authorized to do. The so called "super-user" password, for example, is especially critical because the super-user has all sorts of permissions and has essentially unlimited access to all system resources.

Password security is of course only one component of overall system security, but it is an essential component. Experience has shown that attempts to penetrate remote-access systems have been astonishingly sophisticated.

Remote-access systems are peculiarly vulnerable to penetration by outsiders as there are threats at the remote terminal, along the communications link, as well as at the computer itself. Although the security of a password encryption algorithm is an interesting intellectual and mathematical problem, it is only one tiny facet of a very large problem. In practice, physical security of the computer, communications security of the communications link, and physical control of the computer itself loom as far more important issues. Perhaps most important of all is control over the actions of ex-employees, since they are not under any direct control and they may have intimate knowledge about the system, its resources, and methods of access. Good system security involves realistic evaluation of the risks not only of deliberate attacks but also of casual unauthorized access and accidental disclosure.

---

†UNIX is a Trademark of Bell Laboratories.

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 typicallly about 250,000 words; these words can be checked in about five minutes. Again, a noticeable fraction of any collection of passwords will be found. Improvements and extensions will be (and have been) found by a determined bad guy. Some "good" things to try are:

- The dictionary with the words spelled backwards.
- A list of first names (best obtained from some mailing list). Last names, street names, and city names also work well.
- The above with initial upper-case letters.
- All valid license plate numbers in your state. (This takes about five hours in New Jersey.)
- Room numbers, social security numbers, telephone numbers, and the like.

The authors have conducted experiments to try to determine typical users' habits in the choice of passwords when no constraint is put on their choice. The results were disappointing, except to the bad guy. In a collection of 3,289 passwords gathered from many users over a long period of time;

15 were a single ASCII character;

72 were strings of two ASCII characters;

464 were strings of three ASCII characters;

477 were string of four alphamerics;

706 were five letters, all upper-case or all lower-case;

605 were six letters, all lower-case.

An additional 492 passwords appeared in various available dictionaries, name lists, and the like. A total of 2,831, or 86% of this sample of passwords fell into one of these classes.

There was, of course, considerable overlap between the dictionary results and the character string searches. The dictionary search alone, which required only five minutes to run, produced about one third of the passwords.

Users could be urged (or forced) to use either longer passwords or passwords chosen from a larger character set, or the system could itself choose passwords for the users.

AN ANECDOTE

An entertaining and instructive example is the attempt made at one installation to force users to use less predictable passwords. The users did not choose their own passwords; the system supplied them. The supplied passwords were eight characters long and were taken from the character set consisting of lower-case letters and digits. They were generated by a pseudo-random number generator with only 2^{15} starting values. The time required to search (again on a PDP-11/70) through all character strings of length 8 from a 36-character alphabet is 112 years.

Unfortunately, only 2^{15} of them need be looked at, because that is the number of possible outputs of the random number generator. The bad guy did, in fact, generate and test each of these strings and found every one of the system-generated passwords using a total of only about one minute of machine time.

IMPROVEMENTS TO THE FIRST APPROACH

1. Slower Encryption

Obviously, the first algorithm used was far too fast. The announcement of the DES encryption algorithm [2] by the National Bureau of Standards was timely and fortunate. The DES is, by design, hard to invert, but equally valuable is the fact that it is extremely slow when implemented in software. The DES was implemented and used in the following way: The first eight characters of the user's password are used as a key for the DES; then the algorithm is used to encrypt a constant. Although this constant is zero at the moment, it is easily accessible and can be made installation-dependent. Then the DES algorithm is iterated 25 times and the resulting 64 bits are repacked to become a string of 11 printable characters.

2. Less Predictable Passwords

The password entry program was modified so as to urge the user to use more obscure passwords. If the user enters an alphabetic password (all upper-case or all lower-case) shorter than six characters, or a password from a larger character set shorter than five characters, then the program asks him to enter a longer password. This further reduces the efficacy of key search.

These improvements make it exceedingly difficult to find any individual password. The user is warned of the risks and if he cooperates, he is very safe indeed. On the other hand, he is not prevented from using his spouse's name if he wants to.

3. Salted Passwords

The key search technique is still likely to turn up a few passwords when it is used on a large collection of passwords, and it seemed wise to make this task as difficult as possible. To this end, when a password is first entered, the password program obtains a 12-bit random number (by reading the real-time clock) and appends this to the password typed in by the user. The concatenated string is encrypted and both the 12-bit random quantity (called the *salt*) and the 64-bit result of the encryption are entered into the password file.

When the user later logs in to the system, the 12-bit quantity is extracted from the password file and appended to the typed password. The encrypted result is required, as before, to be the same as the remaining 64 bits in the password file. This modification does not increase the task of finding any individual password, starting from scratch, but now the work of testing a given character string against a large collection of encrypted passwords has been multiplied by 4096 (2^{12}). The reason for this is that there are 4096 encrypted versions of each password and one of them has been picked more or less at random by the system.

With this modification, it is likely that the bad guy can spend days of computer time trying to find a password on a system with hundreds of passwords, and find none at all. More important is the fact that it becomes impractical to prepare an encrypted dictionary in advance. Such an encrypted dictionary could be used to crack new passwords in milliseconds when they appear.

There is a (not inadvertent) side effect of this modification. It becomes nearly impossible to find out whether a person with passwords on two or more systems has used the same password on all of them, unless you already know that.

4. The Threat of the DES Chip

Chips to perform the DES encryption are already commercially available and they are very fast. The use of such a chip speeds up the process of password hunting by three orders of magnitude. To avert this possibility, one of the internal tables of the DES algorithm (in particular, the so-called E-table) is changed in a way that depends on the 12-bit random number. The E-table is inseparably wired into the DES chip, so that the commercial chip cannot be used. Obviously, the bad guy could have his own chip designed and built, but the cost would be unthinkable.

5. A Subtle Point

To login successfully on the UNIX system, it is necessary after dialing in to type a valid user name, and then the correct password for that user name. It is poor design to write the login command in such a way that it tells an interloper when he has typed in a invalid user name. The response to an invalid name should be identical to that for a valid name.

When the slow encryption algorithm was first implemented, the encryption was done only if the user name was valid, because otherwise there was no encrypted password to compare with the supplied password. The result was that the response was delayed by about one-half second if the name was valid, but was immediate if invalid. The bad guy could find out whether a particular user name was valid. The routine was modified to do the encryption in either case.

CONCLUSIONS

On the issue of password security, UNIX is probably better than most systems. The use of encrypted passwords appears reasonably secure in the absence of serious attention of experts in the field.

It is also worth some effort to conceal even the encrypted passwords. Some UNIX systems have instituted what is called an "external security code" that must be typed when dialing into the system, but before logging in. If this code is changed periodically, then someone with an old password will likely be prevented from using it.

Whenever any security procedure is instituted that attempts to deny access to unauthorized persons, it is wise to keep a record of both successful and unsuccessful attempts to get at the secured resource. Just as an out-of-hours visitor to a computer center normally must not only identify himself, but a record is usually also kept of his entry. Just so, it is a wise precaution to make and keep a record of all attempts to log into a remote-access time-sharing system, and certainly all unsuccessful attempts.

Bad guys fall on a spectrum whose one end is someone with ordinary access to a system and whose goal is to find out a particular password (usually that of the super-user) and, at the other end, someone who wishes to collect as much password information as possible from as many systems as possible. Most of the work reported here serves to frustrate the latter type; our experience indicates that the former type of bad guy never was very successful.

We recognize that a time-sharing system must operate in a hostile environment. We did not attempt to hide the security aspects of the operating system, thereby playing the customary make-believe game in which weaknesses of the system are not discussed no matter how apparent. Rather we advertised the password algorithm and invited attack in the belief that this approach would minimize future trouble. The approach has been successful.

References

[1] Ritchie, D.M. and Thompson, K. The UNIX Time-Sharing System. *Comm. ACM* **17** (July 1974), pp. 365-375.

[2] *Proposed Federal Information Processing Data Encryption Standard.* Federal Register (40FR12134), March 17, 1975

[3] Wilkes, M. V. *Time-Sharing Computer Systems.* American Elsevier, New York, (1968).

[4] U. S. Patent Number 2,089,603.

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 t pes 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.