Unix Programmer's Manual
Supplementary Documents 1

Unix Programmer's Manual
Supplementary Documents 1

PS1

Cover design by John Lassetter, Lucasfilm, Ltd.

4.2 BSD edition:

First Printing	July 1984
Second Printing	December 1984
Third Printing	September 1985
Fourth Printing	March 1986

4.3 BSD edition:

First Printing	November 1986
Second Printing	June 1987

UNIX Programmer's Supplementary Documents
Volume 1
(PS1)

4.3 Berkeley Software Distribution
Virtual VAX-11 Version

April, 1986

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

This manual reflects system enhancements made at Berkeley and sponsored in part by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871 monitored by the Naval Electronics Systems Command under contract No. N00039-84-C-0089. The views and conclusions contained in these documents are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the US Government.

UNIX Programmer's Supplementary Documents, Volume 1 (PS1)

4.3 Berkeley Software Distribution, Virtual VAX-11 Version

April, 1986

These two volumes contain documents which supplement the manual pages in *The UNIX† Programmer's Reference Manual* for the Virtual VAX-11 version of the system as distributed by U.C. Berkeley.

Languages in common use (other languages in Programmer's Supplement, volume 2)

General Reference

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

The C Programming Language - Reference Manual

Dennis M. Ritchie

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

This manual is a reprint, with updates to the current C standard, from *The C Programming Language*, by Brian W. Kernighan and Dennis M. Richie, Prentice-Hall, Inc., 1978.

1. Introduction

This manual describes the C language on the DEC PDP-11†, the DEC VAX-11, and the AT&T 3B 20‡. Where differences exist, it concentrates on the VAX, but tries to point out implementation-dependent details. With few execptions, these dependencies follow directly from the underlying properties of the hardware; the various compilers are generally quite compatible.

2. Lexical Conventions

There are six classes of tokens - identifiers, keywords, constants, strings, operators, and other separators. Blanks, tabs, new-lines, and comments (collectively, "white space") as described below are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

2.1. Comments

The characters /* introduce a comment which terminates with the characters */. Comments do not nest.

2.2. Identifiers (Names)

An identifier is a sequence of letters and digits. The first character must be a letter. The underscore (_) counts as a letter. Uppercase and lowercase letters are different. Although there is no limit on the length of a name, only initial characters are significant: at least eight characters of a non-external name, and perhaps fewer for external names. Moreover, some implementations may collapse case distinctions for external names. The external name sizes include:

PDP-11	7 characters, 2 cases
VAX-11	>100 characters, 2 cases
AT&T 3B 20	>100 characters, 2 cases

2.3. Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise:

---

† DEC PDP-11, and DEC VAX-11 are trademarks of Digital Equipment Corporation.

‡ 3B 20 is a trademark of AT&T.

auto	**do**	**for**	**return**	**typedef**
break	**double**	**goto**	**short**	**union**
case	**else**	**if**	**sizeof**	**unsigned**
char	**enum**	**int**	**static**	**void**
continue	**external**	**long**	**struct**	**while**
default	**float**	**register**	**switch**	

Some implementations also reserve the words **fortran, asm, gfloat, hfloat** and **quad**

2.4. Constants

There are several kinds of constants. Each has a type; an introduction to types is given in "NAMES." Hardware characteristics that affect sizes are summarized in "Hardware Characteristics" under "LEXICAL CONVENTIONS."

2.4.1. Integer Constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with **0** (digit zero). An octal constant consists of the digits **0** through **7** only. A sequence of digits preceded by **0x** or **0X** (digit zero) is taken to be a hexadecimal integer. The hexadecimal digits include **a** or **A** through **f** or **F** with values 10 through 15. Otherwise, the integer constant is taken to be decimal. A decimal constant whose value exceeds the largest signed machine integer is taken to be **long**; an octal or hex constant which exceeds the largest unsigned machine integer is likewise taken to be **long**. Otherwise, integer constants are **int**.

2.4.2. Explicit Long Constants

A decimal, octal, or hexadecimal integer constant immediately followed by **l** (letter ell) or **L** is a long constant. As discussed below, on some machines integer and long values may be considered identical.

2.4.3. Character Constants

A character constant is a character enclosed in single quotes, as in **'x'**. The value of a character constant is the numerical value of the character in the machine's character set.

Certain nongraphic characters, the single quote (') and the backslash (\), may be represented according to the following table of escape sequences:

new-line	NL (LF)	\n
horizontal tab	HT	\t
vertical tab	VT	\v
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
backslash	\	\\
single quote	'	\'
bit pattern	*ddd*	*ddd*

The escape *\ddd* consists of the backslash followed by 1, 2, or 3 octal digits which are taken to specify the value of the desired character. A special case of this construction is **\0** (not followed by a digit), which indicates the character **NUL**. If the character following a backslash is not one of those specified, the behavior is undefined. A new-line character is illegal in a character constant. The type of a character constant is **int**.

2.4.4. Floating Constants

A floating constant consists of an integer part, a decimal point, a fraction part, an **e** or **E**, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing. Either the decimal point or the **e** and the exponent (not both) may be missing. Every floating constant has type **double.**

2.4.5. Enumeration Constants

Names declared as enumerators (see "Structure, Union, and Enumeration Declarations" under "DECLARATIONS") have type **int.**

2.5. Strings

A string is a sequence of characters surrounded by double quotes, as in "...". A string has type "array of **char**" and storage class **static** (see "NAMES") and is initialized with the given characters. The compiler places a null byte (**\0**) at the end of each string so that programs which scan the string can find its end. In a string, the double quote character (") must be preceded by a \; in addition, the same escapes as described for character constants may be used.

A \ and the immediately following new-line are ignored. All strings, even when written identically, are distinct.

2.6. Hardware Characteristics

The following figure summarize certain hardware properties that vary from machine to machine.

	DEC PDP-11 (ASCII)	**DEC VAX-11** (ASCII)	**AT&T 3B** (ASCII)
char	8 bits	8 bits	8bits
int	16	32	32
short	16	16	16
long	32	32	32
float	32	32	32
double	64	64	64
float range	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$
double range	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$	$\pm 10^{\pm 308}$

3. Syntax Notation

Syntactic categories are indicated by *italic* type and literal words and characters in **bold** type. Alternative categories are listed on separate lines. An optional terminal or nonterminal symbol is indicated by the subscript "opt," so that

$$\{\ expression_{opt}\ \}$$

indicates an optional expression enclosed in braces. The syntax is summarized in "SYNTAX SUMMARY".

4. Names

The C language bases the interpretation of an identifier upon two attributes of the identifier – its *storage class* and its *type.* The storage class determines the location and lifetime of the storage associated with an identifier; the type determines the meaning of the values found in the identifier's storage.

4.1. Storage Class

There are four declarable storage classes: Automatic Static External Register.

Automatic variables are local to each invocation of a block (see "Compound Statement or Block" in "STATEMENTS") and are discarded upon exit from the block. Static variables are local to a block but retain their values upon reentry to a block even after control has left the block. External variables exist and retain their values throughout the execution of the entire program and may be used for communication between functions, even separately compiled functions. Register variables are (if possible) stored in the fast registers of the machine; like automatic variables, they are local to each block and disappear on exit from the block.

4.2. Type

The C language supports several fundamental types of objects. Objects declared as characters (**char**) are large enough to store any member of the implementation's character set. If a genuine character from that character set is stored in a **char** variable, its value is equivalent to the integer code for that character. Other quantities may be stored into character variables, but the implementation is machine dependent. In particular, **char** may be signed or unsigned by default.

Up to three sizes of integer, declared **short int**, **int**, and **long int**, are available. Longer integers provide no less storage than shorter ones, but the implementation may make either short integers or long integers, or both, equivalent to plain integers. "Plain" integers have the natural size suggested by the host machine architecture. The other sizes are provided to meet special needs.

The properties of **enum** types (see "Structure, Union, and Enumeration Declarations" under "DECLARATIONS") are identical to those of some integer types. The implementation may use the range of values to determine how to allocate storage.

Unsigned integers, declared **unsigned,** obey the laws of arithmetic modulo 2^n where n is the number of bits in the representation. (On the PDP-11, unsigned long quantities are not supported.)

Single-precision floating point (**float**) and double precision floating point (**double**) may be synonymous in some implementations.

Because objects of the foregoing types can usefully be interpreted as numbers, they will be referred to as *arithmetic* types. **Char**, **int** of all sizes whether **unsigned** or not, and **enum** will collectively be called *integral* types. The **float** and **double** types will collectively be called *floating* types.

The **void** type specifies an empty set of values. It is used as the type returned by functions that generate no value.

Besides the fundamental arithmetic types, there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways: *Arrays* of objects of most types *Functions* which return objects of a given type *Pointers* to objects of a given type *Structures* containing a sequence of objects of various types *Unions* capable of containing any one of several objects of various types.

In general these methods of constructing objects can be applied recursively.

5. Objects and Lvalues

An *object* is a manipulatable region of storage. An *lvalue* is an expression referring to an object. An obvious example of an lvalue expression is an identifier. There are operators which yield lvalues: for example, if **E** is an expression of pointer type, then ***E** is an lvalue expression referring to the object to which **E** points. The name "lvalue" comes from the assignment expression **E1 = E2** in which the left operand **E1** must be an lvalue expression. The discussion of each operator below indicates whether it expects lvalue operands and whether it yields an lvalue.

6. Conversions

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This part explains the result to be expected from such conversions. The conversions demanded by most ordinary operators are summarized under "Arithmetic

Conversions." The summary will be supplemented as required by the discussion of each operator.

6.1. Characters and Integers

A character or a short integer may be used wherever an integer may be used. In all cases the value is converted to an integer. Conversion of a shorter integer to a longer preserves sign. Whether or not sign-extension occurs for characters is machine dependent, but it is guaranteed that a member of the standard character set is non-negative. Of the machines treated here, only the PDP-11 and VAX-11 sign-extend. On these machines, **char** variables range in value from −128 to 127. The more explicit type **unsigned char** forces the values to range from 0 to 255.

On machines that treat characters as signed, the characters of the ASCII set are all non-negative. However, a character constant specified with an octal escape suffers sign extension and may appear negative; for example, '\377' has the value **−1**.

When a longer integer is converted to a shorter integer or to a **char**, it is truncated on the left. Excess bits are simply discarded.

6.2. Float and Double

All floating arithmetic in C is carried out in double precision. Whenever a **float** appears in an expression it is lengthened to **double** by zero padding its fraction. When a **double** must be converted to **float**, for example by an assignment, the **double** is rounded before truncation to **float** length. This result is undefined if it cannot be represented as a float. On the VAX, the compiler can be directed to use single percision for expressions containing only float and interger operands.

6.3. Floating and Integral

Conversions of floating values to integral type are rather machine dependent. In particular, the direction of truncation of negative numbers varies. The result is undefined if it will not fit in the space provided.

Conversions of integral values to floating type are well behaved. Some loss of accuracy occurs if the destination lacks sufficient bits.

6.4. Pointers and Integers

An expression of integral type may be added to or subtracted from a pointer; in such a case, the first is converted as specified in the discussion of the addition operator. Two pointers to objects of the same type may be subtracted; in this case, the result is converted to an integer as specified in the discussion of the subtraction operator.

6.5. Unsigned

Whenever an unsigned integer and a plain integer are combined, the plain integer is converted to unsigned and the result is unsigned. The value is the least unsigned integer congruent to the signed integer (modulo $2^{wordsize}$). In a 2's complement representation, this conversion is conceptual; and there is no actual change in the bit pattern.

When an unsigned **short** integer is converted to **long**, the value of the result is the same numerically as that of the unsigned integer. Thus the conversion amounts to padding with zeros on the left.

6.6. Arithmetic Conversions

A great many operators cause conversions and yield result types in a similar way. This pattern will be called the "usual arithmetic conversions." First, any operands of type **char** or **short** are converted to **int**, and any operands of type **unsigned char** or **unsigned short** are converted to **unsigned int**. Then, if either operand is **double**, the other is converted to **double** and that is the type of the result. Otherwise, if either operand is **unsigned long**, the other is converted to **unsigned long** and that is the type of the result. Otherwise, if either operand is **long**, the other is converted to **long** and that is the type of the result. Otherwise, if one operand is **long**, and the other is **unsigned int**, they are both

converted to **unsigned long** and that is the type of the result. Otherwise, if either operand is **unsigned,** the other is converted to **unsigned** and that is the type of the result. Otherwise, both operands must be **int**, and that is the type of the result.

6.7. Void

The (nonexistent) value of a **void** object may not be used in any way, and neither explicit nor implicit conversion may be applied. Because a void expression denotes a nonexistent value, such an expression may be used only as an expression statement (see "Expression Statement" under "STATEMENTS") or as the left operand of a comma expression (see "Comma Operator" under "EXPRESSIONS").

An expression may be converted to type **void** by use of a cast. For example, this makes explicit the discarding of the value of a function call used as an expression statement.

7. Expressions

The precedence of expression operators is the same as the order of the major subsections of this section, highest precedence first. Thus, for example, the expressions referred to as the operands of + (see "Additive Operators") are those expressions defined under "Primary Expressions", "Unary Operators", and "Multiplicative Operators". Within each subpart, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The precedence and associativity of all the expression operators are summarized in the grammar of "SYNTAX SUMMARY".

Otherwise, the order of evaluation of expressions is undefined. In particular, the compiler considers itself free to compute subexpressions in the order it believes most efficient even if the subexpressions involve side effects. The order in which subexpression evaluation takes place is unspecified. Expressions involving a commutative and associative operator (*, +, &, |, ^) may be rearranged arbitrarily even in the presence of parentheses; to force a particular order of evaluation, an explicit temporary must be used.

The handling of overflow and divide check in expression evaluation is undefined. Most existing implementations of C ignore integer overflows; treatment of division by 0 and all floating-point exceptions varies between machines and is usually adjustable by a library function.

7.1. Primary Expressions

Primary expressions involving ., ->, subscripting, and function calls group left to right.

primary-expression:
- *identifier*
- *constant*
- *string*
- *(expression)*
- *primary-expression [expression]*
- *primary-expression (expression-list*$_{opt}$ *)*
- *primary-expression . identifier*
- *primary-expression -> identifier*

expression-list:
- *expression*
- *expression-list , expression*

An identifier is a primary expression provided it has been suitably declared as discussed below. Its type is specified by its declaration. If the type of the identifier is "array of ...", then the value of the identifier expression is a pointer to the first object in the array; and the type of the expression is "pointer to ...". Moreover, an array identifier is not an lvalue expression. Likewise, an identifier which is declared "function returning ...", when used except in the function-name position of a call,

is converted to "pointer to function returning ...".

A constant is a primary expression. Its type may be **int**, **long**, or **double** depending on its form. Character constants have type **int** and floating constants have type **double**.

A string is a primary expression. Its type is originally "array of **char**", but following the same rule given above for identifiers, this is modified to "pointer to **char**" and the result is a pointer to the first character in the string. (There is an exception in certain initializers; see "Initialization" under "DECLARATIONS.")

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

A primary expression followed by an expression in square brackets is a primary expression. The intuitive meaning is that of a subscript. Usually, the primary expression has type "pointer to ...", the subscript expression is **int**, and the type of the result is "...". The expression **E1[E2]** is identical (by definition) to ***((E1)+E2)**. All the clues needed to understand this notation are contained in this subpart together with the discussions in "Unary Operators" and "Additive Operators" on identifiers, * and + respectively. The implications are summarized under "Arrays, Pointers, and Subscripting" under "TYPES REVISITED."

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. The primary expression must be of type "function returning ...," and the result of the function call is of type "...". As indicated below, a hitherto unseen identifier followed immediately by a left parenthesis is contextually declared to represent a function returning an integer; thus in the most common case, integer-valued functions need not be declared.

Any actual arguments of type **float** are converted to **double** before the call. Any of type **char** or **short** are converted to **int**. Array names are converted to pointers. No other conversions are performed automatically; in particular, the compiler does not compare the types of actual arguments with those of formal arguments. If conversion is needed, use a cast; see "Unary Operators" and "Type Names" under "DECLARATIONS."

In preparing for the call to a function, a copy is made of each actual parameter. Thus, all argument passing in C is strictly by value. A function may change the values of its formal parameters, but these changes cannot affect the values of the actual parameters. It is possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points. An array name is a pointer expression. The order of evaluation of arguments is undefined by the language; take note that the various compilers differ. Recursive calls to any function are permitted.

A primary expression followed by a dot followed by an identifier is an expression. The first expression must be a structure or a union, and the identifier must name a member of the structure or union. The value is the named member of the structure or union, and it is an lvalue if the first expression is an lvalue.

A primary expression followed by an arrow (built from - and >) followed by an identifier is an expression. The first expression must be a pointer to a structure or a union and the identifier must name a member of that structure or union. The result is an lvalue referring to the named member of the structure or union to which the pointer expression points. Thus the expression **E1->MOS** is the same as **(*E1).MOS**. Structures and unions are discussed in "Structure, Union, and Enumeration Declarations" under "DECLARATIONS."

7.2. Unary Operators

Expressions with unary operators group right to left.

unary-expression:
** expression*
& lvalue
- expression
! expression
~ expression
++ lvalue
--lvalue
lvalue ++
lvalue --
(type-name) expression
sizeof *expression*
sizeof *(type-name)*

The unary * operator means *indirection* ; the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points. If the type of the expression is "pointer to ...," the type of the result is "...".

The result of the unary **&** operator is a pointer to the object referred to by the lvalue. If the type of the lvalue is "...", the type of the result is "pointer to ...".

The result of the unary - operator is the negative of its operand. The usual arithmetic conversions are performed. The negative of an unsigned quantity is computed by subtracting its value from 2^n where n is the number of bits in the corresponding signed type.

There is no unary + operator.

The result of the logical negation operator **!** is one if the value of its operand is zero, zero if the value of its operand is nonzero. The type of the result is **int**. It is applicable to any arithmetic type or to pointers.

The ~ operator yields the one's complement of its operand. The usual arithmetic conversions are performed. The type of the operand must be integral.

The object referred to by the lvalue operand of prefix ++ is incremented. The value is the new value of the operand but is not an lvalue. The expression **++x** is equivalent to **x=x+1**. See the discussions "Additive Operators" and "Assignment Operators" for information on conversions.

The lvalue operand of prefix -- is decremented analogously to the prefix ++ operator.

When postfix ++ is applied to an lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is incremented in the same manner as for the prefix ++ operator. The type of the result is the same as the type of the lvalue expression.

When postfix -- is applied to an lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is decremented in the manner as for the prefix -- operator. The type of the result is the same as the type of the lvalue expression.

An expression preceded by the parenthesized name of a data type causes conversion of the value of the expression to the named type. This construction is called a *cast*. Type names are described in "Type Names" under "Declarations."

The **sizeof** operator yields the size in bytes of its operand. (A *byte* is undefined by the language except in terms of the value of **sizeof**. However, in all existing implementations, a byte is the space required to hold a **char**.) When applied to an array, the result is the total number of bytes in the array. The size is determined from the declarations of the objects in the expression. This expression is semantically an **unsigned** constant and may be used anywhere a constant is required. Its major use is in communication with routines like storage allocators and I/O systems.

The **sizeof** operator may also be applied to a parenthesized type name. In that case it yields the size in bytes of an object of the indicated type.

The construction **sizeof(***type* **)** is taken to be a unit, so the expression **sizeof(***type***)-2** is the same as **(sizeof(***type***))-2**.

7.3. Multiplicative Operators

The multiplicative operators *, /, and % group left to right. The usual arithmetic conversions are performed.

multiplicative expression:
*expression * expression*
expression / expression
expression % expression

The binary * operator indicates multiplication. The * operator is associative, and expressions with several multiplications at the same level may be rearranged by the compiler. The binary / operator indicates division.

The binary % operator yields the remainder from the division of the first expression by the second. The operands must be integral.

When positive integers are divided, truncation is toward 0; but the form of truncation is machine-dependent if either operand is negative. On all machines covered by this manual, the remainder has the same sign as the dividend. It is always true that **(a/b)*b** + **a%b** is equal to **a** (if **b** is not 0).

7.4. Additive Operators

The additive operators + and – group left to right. The usual arithmetic conversions are performed. There are some additional type possibilities for each operator.

additive-expression:
expression + expression
expression – expression

The result of the + operator is the sum of the operands. A pointer to an object in an array and a value of any integral type may be added. The latter is in all cases converted to an address offset by multiplying it by the length of the object to which the pointer points. The result is a pointer of the same type as the original pointer which points to another object in the same array, appropriately offset from the original object. Thus if **P** is a pointer to an object in an array, the expression **P+1** is a pointer to the next object in the array. No further type combinations are allowed for pointers.

The + operator is associative, and expressions with several additions at the same level may be rearranged by the compiler.

The result of the – operator is the difference of the operands. The usual arithmetic conversions are performed. Additionally, a value of any integral type may be subtracted from a pointer, and then the same conversions for addition apply.

If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an **int** representing the number of objects separating the pointed-to objects. This conversion will in general give unexpected results unless the pointers point to objects in the same array, since pointers, even to objects of the same type, do not necessarily differ by a multiple of the object length.

7.5. Shift Operators

The shift operators << and >> group left to right. Both perform the usual arithmetic conversions on their operands, each of which must be integral. Then the right operand is converted to **int**; the type of the result is that of the left operand. The result is undefined if the right operand is negative or greater than or equal to the length of the object in bits. On the VAX a negative right operand is interpreted as reversing the direction of the shift.

shift-expression:
expression << *expression*
expression >> *expression*

The value of **E1<<E2** is **E1** (interpreted as a bit pattern) left-shifted **E2** bits. Vacated bits are 0 filled. The value of **E1>>E2** is **E1** right-shifted **E2** bit positions. The right shift is guaranteed to be logical (0 fill) if **E1** is **unsigned**; otherwise, it may be arithmetic.

7.6. Relational Operators

The relational operators group left to right.

relational-expression:
expression < *expression*
expression > *expression*
expression <= *expression*
expression >= *expression*

The operators < (less than), > (greater than), <= (less than or equal to), and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is **int**. The usual arithmetic conversions are performed. Two pointers may be compared; the result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is portable only when the pointers point to objects in the same array.

7.7. Equality Operators

equality-expression:
expression == *expression*
expression != *expression*

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus **a<b == c<d** is 1 whenever **a<b** and **c<d** have the same truth value).

A pointer may be compared to an integer only if the integer is the constant 0. A pointer to which 0 has been assigned is guaranteed not to point to any object and will appear to be equal to 0. In conventional usage, such a pointer is considered to be null.

7.8. Bitwise AND Operator

and-expression:
expression & *expression*

The & operator is associative, and expressions involving & may be rearranged. The usual arithmetic conversions are performed. The result is the bitwise AND function of the operands. The operator applies only to integral operands.

7.9. Bitwise Exclusive OR Operator

exclusive-or-expression:
expression ^ *expression*

The ^ operator is associative, and expressions involving ^ may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of the operands. The operator applies only to integral operands.

7.10. Bitwise Inclusive OR Operator

inclusive-or-expression:
expression | *expression*

The | operator is associative, and expressions involving | may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of its operands. The operator applies only to integral operands.

7.11. Logical AND Operator

logical-and-expression:
expression && expression

The **&&** operator groups left to right. It returns 1 if both its operands evaluate to nonzero, 0 otherwise. Unlike **&**, **&&** guarantees left to right evaluation; moreover, the second operand is not evaluated if the first operand is 0.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always **int**.

7.12. Logical OR Operator

logical-or-expression:
expression || *expression*

The || operator groups left to right. It returns 1 if either of its operands evaluates to nonzero, 0 otherwise. Unlike |, || guarantees left to right evaluation; moreover, the second operand is not evaluated if the value of the first operand is nonzero.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always **int**.

7.13. Conditional Operator

conditional-expression:
expression ? expression : expression

Conditional expressions group right to left. The first expression is evaluated; and if it is nonzero, the result is the value of the second expression, otherwise that of third expression. If possible, the usual arithmetic conversions are performed to bring the second and third expressions to a common type. If both are structures or unions of the same type, the result has the type of the structure or union. If both pointers are of the same type, the result has the common type. Otherwise, one must be a pointer and the other the constant 0, and the result has the type of the pointer. Only one of the second and third expressions is evaluated.

7.14. Assignment Operators

There are a number of assignment operators, all of which group right to left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place. The two parts of a compound assignment operator are separate tokens.

assignment-expression:
lvalue = expression
lvalue += expression
lvalue −= expression
*lvalue *= expression*
lvalue /= expression
lvalue %= expression
lvalue >>= expression
lvalue <<= expression
lvalue &= expression
lvalue ^= expression
lvalue |= expression

In the simple assignment with =, the value of the expression replaces that of the object referred to by the lvalue. If both operands have arithmetic type, the right operand is converted to the type of the left preparatory to the assignment. Second, both operands may be structures or unions of the same type. Finally, if the left operand is a pointer, the right operand must in general be a pointer of the same type. However, the constant 0 may be assigned to a pointer; it is guaranteed that this value will produce a null pointer distinguishable from a pointer to any object.

The behavior of an expression of the form **E1** *op* = **E2** may be inferred by taking it as equivalent to **E1** = **E1** *op* (**E2**); however, **E1** is evaluated only once. In += and −=, the left operand may be a pointer; in which case, the (integral) right operand is converted as explained in "Additive Operators." All right operands and all nonpointer left operands must have arithmetic type.

7.15. Comma Operator

comma-expression:
expression , expression

A pair of expressions separated by a comma is evaluated left to right, and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left to right. In contexts where comma is given a special meaning, e.g., in lists of actual arguments to functions (see "Primary Expressions") and lists of initializers (see "Initialization" under "DECLARATIONS"), the comma operator as described in this subpart can only appear in parentheses. For example,

f(a, (t=3, t+2), c)

has three arguments, the second of which has the value 5.

8. Declarations

Declarations are used to specify the interpretation which C gives to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations have the form

declaration:
decl-specifiers declarator-list$_{opt}$;

The declarators in the declarator-list contain the identifiers being declared. The decl-specifiers consist of a sequence of type and storage class specifiers.

decl-specifiers:
type-specifier decl-specifiers$_{opt}$
sc-specifier decl-specifiers$_{opt}$

The list must be self-consistent in a way described below.

8.1. Storage Class Specifiers

The sc-specifiers are:

sc-specifier:
auto
static
extern
register
typedef

The **typedef** specifier does not reserve storage and is called a "storage class specifier" only for syntactic convenience. See "Typedef" for more information. The meanings of the various storage classes were discussed in "Names."

The **auto**, **static**, and **register** declarations also serve as definitions in that they cause an appropriate amount of storage to be reserved. In the **extern** case, there must be an external definition (see "External Definitions") for the given identifiers somewhere outside the function in which they are declared.

A **register** declaration is best thought of as an **auto** declaration, together with a hint to the compiler that the variables declared will be heavily used. Only the first few such declarations in each function are effective. Moreover, only variables of certain types will be stored in registers; on the PDP-11, they are **int** or pointer. One other restriction applies to register variables: the address-of operator **&** cannot be applied to them. Smaller, faster programs can be expected if register declarations are used appropriately, but future improvements in code generation may render them unnecessary.

At most, one sc-specifier may be given in a declaration. If the sc-specifier is missing from a declaration, it is taken to be **auto** inside a function, **extern** outside. Exception: functions are never automatic.

8.2. Type Specifiers

The type-specifiers are

type-specifier:
struct-or-union-specifier
typedef-name
enum-specifier
basic-type-specifier:
basic-type
basic-type basic-type-specifiers
basic-type:
char
short
int
long
unsigned
float
double
void

At most one of the words **long** or **short** may be specified in conjunction with **int**; the meaning is the same as if **int** were not mentioned. The word **long** may be specified in conjunction with **float**; the meaning is the same as **double**. The word **unsigned** may be specified alone, or in conjunction with **int** or any of its short or long varieties, or with **char**.

Otherwise, at most on type-specifier may be given in a declaration. In particular, adjectival use of **long**, **short**, or **unsigned** is not permitted with **typedef** names. If the type-specifier is missing from a

declaration, it is taken to be **int**.

Specifiers for structures, unions, and enumerations are discussed in "Structure, Union, and Enumeration Declarations." Declarations with **typedef** names are discussed in "Typedef."

8.3. Declarators

The declarator-list appearing in a declaration is a comma-separated sequence of declarators, each of which may have an initializer.

declarator-list:
init-declarator
init-declarator , declarator-list

init-declarator:
declarator initializer$_{opt}$

Initializers are discussed in "Initialization". The specifiers in the declaration indicate the type and storage class of the objects to which the declarators refer. Declarators have the syntax:

declarator:
identifier
(declarator)
** declarator*
declarator ()
declarator [constant-expression$_{opt}$ *]*

The grouping is the same as in expressions.

8.4. Meaning of Declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type and storage class.

Each declarator contains exactly one identifier; it is this identifier that is declared. If an unadorned identifier appears as a declarator, then it has the type indicated by the specifier heading the declaration.

A declarator in parentheses is identical to the unadorned declarator, but the binding of complex declarators may be altered by parentheses. See the examples below.

Now imagine a declaration

T D1

where **T** is a type-specifier (like **int**, etc.) and **D1** is a declarator. Suppose this declaration makes the identifier have type "... **T** ," where the "..." is empty if **D1** is just a plain identifier (so that the type of **x** in **'int x"** is just **int**). Then if **D1** has the form

***D**

the type of the contained identifier is "... pointer to **T** ."

If **D1** has the form

D( )

then the contained identifier has the type "... function returning **T**."

If **D1** has the form

D [*constant-expression* **]**

or

D[]

then the contained identifier has type "... array of T." In the first case, the constant expression is an expression whose value is determinable at compile time , whose type is **int**, and whose value is positive. (Constant expressions are defined precisely in "Constant Expressions.") When several "array of" specifications are adjacent, a multidimensional array is created; the constant expressions which specify the bounds of the arrays may be missing only for the first member of the sequence. This elision is useful when the array is external and the actual definition, which allocates storage, is given elsewhere. The first constant expression may also be omitted when the declarator is followed by initialization. In this case the size is calculated from the number of initial elements supplied.

An array may be constructed from one of the basic types, from a pointer, from a structure or union, or from another array (to generate a multidimensional array).

Not all the possibilities allowed by the syntax above are actually permitted. The restrictions are as follows: functions may not return arrays or functions although they may return pointers; there are no arrays of functions although there may be arrays of pointers to functions. Likewise, a structure or union may not contain a function; but it may contain a pointer to a function.

As an example, the declaration

int i, *ip, f(), *fip(), (*pfi)();

declares an integer **i**, a pointer **ip** to an integer, a function **f** returning an integer, a function **fip** returning a pointer to an integer, and a pointer **pfi** to a function which returns an integer. It is especially useful to compare the last two. The binding of ***fip()** is ***(fip())**. The declaration suggests, and the same construction in an expression requires, the calling of a function **fip**. Using indirection through the (pointer) result to yield an integer. In the declarator **(*pfi)()**, the extra parentheses are necessary, as they are also in an expression, to indicate that indirection through a pointer to a function yields a function, which is then called; it returns an integer.

As another example,

float fa[17], *afp[17];

declares an array of **float** numbers and an array of pointers to **float** numbers. Finally,

static int x3d[3][5][7];

declares a static 3-dimensional array of integers, with rank 3×5×7. In complete detail, **x3d** is an array of three items; each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions **x3d**, **x3d[i]**, **x3d[i][j]**, **x3d[i][j][k]** may reasonably appear in an expression. The first three have type "array" and the last has type **int**.

8.5. Structure and Union Declarations

A structure is an object consisting of a sequence of named members. Each member may have any type. A union is an object which may, at a given time, contain any one of several members. Structure and union specifiers have the same form.

struct-or-union-specifier:
struct-or-union { *struct-decl-list* }
struct-or-union identifier { *struct-decl-list* }
struct-or-union identifier

struct-or-union:
struct
union

The struct-decl-list is a sequence of declarations for the members of the structure or union:

struct-decl-list:
struct-declaration
struct-declaration struct-decl-list

struct-declaration:
type-specifier struct-declarator-list ;

struct-declarator-list:
struct-declarator
struct-declarator , struct-declarator-list

In the usual case, a struct-declarator is just a declarator for a member of a structure or union. A structure member may also consist of a specified number of bits. Such a member is also called a *field ;* its length, a non-negative constant expression, is set off from the field name by a colon.

struct-declarator:
declarator
declarator : constant-expression
: constant-expression

Within a structure, the objects declared have addresses which increase as the declarations are read left to right. Each nonfield member of a structure begins on an addressing boundary appropriate to its type; therefore, there may be unnamed holes in a structure. Field members are packed into machine integers; they do not straddle words. A field which does not fit into the space remaining in a word is put into the next word. No field may be wider than a word.

Fields are assigned right to left on the PDP-11 and VAX-11, left to right on the 3B 20.

A struct-declarator with no declarator, only a colon and a width, indicates an unnamed field useful for padding to conform to externally-imposed layouts. As a special case, a field with a width of 0 specifies alignment of the next field at an implementation dependant boundary.

The language does not restrict the types of things that are declared as fields, but implementations are not required to support any but integer fields. Moreover, even **int** fields may be considered to be unsigned. On the PDP-11, fields are not signed and have only integer values; on the VAX-11, fields declared with **int** are treated as containing a sign. For these reasons, it is strongly recommended that fields be declared as **unsigned**. In all implementations, there are no arrays of fields, and the address-of operator **&** may not be applied to them, so that there are no pointers to fields.

A union may be thought of as a structure all of whose members begin at offset 0 and whose size is sufficient to contain any of its members. At most, one of the members can be stored in a union at any time.

A structure or union specifier of the second form, that is, one of

struct *identifier* { *struct-decl-list* }
union *identifier* { *struct-decl-list* }

declares the identifier to be the *structure tag* (or union tag) of the structure specified by the list. A subsequent declaration may then use the third form of specifier, one of

struct *identifier*
union *identifier*

Structure tags allow definition of self-referential structures. Structure tags also permit the long part of the declaration to be given once and used several times. It is illegal to declare a structure or union which contains an instance of itself, but a structure or union may contain a pointer to an instance of itself.

The third form of a structure or union specifier may be used prior to a declaration which gives the complete specification of the structure or union in situations in which the size of the structure or union is unnecessary. The size is unnecessary in two situations: when a pointer to a structure or union is being declared and when a **typedef** name is declared to be a synonym for a structure or union. This, for example, allows the declaration of a pair of structures which contain pointers to each other.

The names of members and tags do not conflict with each other or with ordinary variables. A particular name may not be used twice in the same structure, but the same name may be used in several different structures in the same scope.

A simple but important example of a structure declaration is the following binary tree structure:

```
struct tnode
{
        char tword[20];
        int count;
        struct tnode *left;
        struct tnode *right;
};
```

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration has been given, the declaration

```
struct tnode s, *sp;
```

declares **s** to be a structure of the given sort and **sp** to be a pointer to a structure of the given sort. With these declarations, the expression

```
sp->count
```

refers to the **count** field of the structure to which **sp** points;

```
s.left
```

refers to the left subtree pointer of the structure **s**; and

```
s.right->tword[0]
```

refers to the first character of the **tword** member of the right subtree of **s**.

8.6. Enumeration Declarations

Enumeration variables and constants have integral type.

enum-specifier:
    enum { *enum-list* }
    enum *identifier* { *enum-list* }
    enum *identifier*

enum-list:
    enumerator
    enum-list , *enumerator*

enumerator:
    identifier
    identifier = *constant-expression*

The identifiers in an enum-list are declared as constants and may appear wherever constants are required. If no enumerators with = appear, then the values of the corresponding constants begin at 0

and increase by 1 as the declaration is read from left to right. An enumerator with = gives the associated identifier the value indicated; subsequent identifiers continue the progression from the assigned value.

The names of enumerators in the same scope must all be distinct from each other and from those of ordinary variables.

The role of the identifier in the enum-specifier is entirely analogous to that of the structure tag in a struct-specifier; it names a particular enumeration. For example,

```
enum color { chartreuse, burgundy, claret=20, winedark };
...
enum color **cp, col;
...
col = claret;
cp = &col;
...
if (**cp == burgundy) ...
```

makes **color** the enumeration-tag of a type describing various colors, and then declares **cp** as a pointer to an object of that type, and **col** as an object of that type. The possible values are drawn from the set {0,1,20,21}.

8.7. Initialization

A declarator may specify an initial value for the identifier being declared. The initializer is preceded by = and consists of an expression or a list of values nested in braces.

initializer:
= *expression*
= { *initializer-list* }
= { *initializer-list* , }

initializer-list:
expression
initializer-list , *initializer-list*
{ *initializer-list* }
{ *initializer-list* , }

All the expressions in an initializer for a static or external variable must be constant expressions, which are described in "CONSTANT EXPRESSIONS", or expressions which reduce to the address of a previously declared variable, possibly offset by a constant expression. Automatic or register variables may be initialized by arbitrary expressions involving constants and previously declared variables and functions.

Static and external variables that are not initialized are guaranteed to start off as zero. Automatic and register variables that are not initialized are guaranteed to start off as garbage.

When an initializer applies to a *scalar* (a pointer or an object of arithmetic type), it consists of a single expression, perhaps in braces. The initial value of the object is taken from the expression; the same conversions as for assignment are performed.

When the declared variable is an *aggregate* (a structure or array), the initializer consists of a brace-enclosed, comma-separated list of initializers for the members of the aggregate written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the aggregate. If there are fewer initializers in the list than there are members of the aggregate, then the aggregate is padded with zeros. It is not permitted to initialize unions or automatic aggregates.

Braces may in some cases be omitted. If the initializer begins with a left brace, then the succeeding comma-separated list of initializers initializes the members of the aggregate; it is erroneous

for there to be more initializers than members. If, however, the initializer does not begin with a left brace, then only enough elements from the list are taken to account for the members of the aggregate; any remaining members are left to initialize the next member of the aggregate of which the current aggregate is a part.

A final abbreviation allows a **char** array to be initialized by a string. In this case successive characters of the string initialize the members of the array.

For example,

```
int x[] = { 1, 3, 5 };
```

declares and initializes **x** as a one-dimensional array which has three members, since no size was specified and there are three initializers.

```
float y[4][3] =
{
        { 1, 3, 5 },
        { 2, 4, 6 },
        { 3, 5, 7 },
};
```

is a completely-bracketed initialization: 1, 3, and 5 initialize the first row of the array **y[0]**, namely **y[0][0]**, **y[0][1]**, and **y[0][2]**. Likewise, the next two lines initialize **y[1]** and **y[2]**. The initializer ends early and therefore **y[3]** is initialized with 0. Precisely, the same effect could have been achieved by

```
float y[4][3] =
{
        1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for **y** begins with a left brace but that for **y[0]** does not; therefore, three elements from the list are used. Likewise, the next three are taken successively for **y[1]** and **y[2]**. Also,

```
float y[4][3] =
{
        { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of **y** (regarded as a two-dimensional array) and leaves the rest 0.

Finally,

```
char msg[] = "Syntax error on line %s\n";
```

shows a character array whose members are initialized with a string.

8.8. Type Names

In two contexts (to specify type conversions explicitly by means of a cast and as an argument of **sizeof**), it is desired to supply the name of a data type. This is accomplished using a "type name", which in essence is a declaration for an object of that type which omits the name of the object.

type-name:
    type-specifier abstract-declarator

abstract-declarator:
empty
(abstract-declarator)
** abstract-declarator*
abstract-declarator ()
abstract-declarator [*constant-expression*$_{opt}$]

To avoid ambiguity, in the construction

(abstract-declarator)

the abstract-declarator is required to be nonempty. Under this restriction, it is possible to identify uniquely the location in the abstract-declarator where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. For example,

```
int
int *
int *[3]
int (*)[3]
int *()
int (*)()
int (*[3])()
```

name respectively the types "integer," "pointer to integer," "array of three pointers to integers," "pointer to an array of three integers," "function returning pointer to integer," "pointer to function returning an integer," and "array of three pointers to functions returning an integer."

8.9. Typedef

Declarations whose "storage class" is **typedef** do not define storage but instead define identifiers which can be used later as if they were type keywords naming fundamental or derived types.

typedef-name:
identifier

Within the scope of a declaration involving **typedef**, each identifier appearing as part of any declarator therein becomes syntactically equivalent to the type keyword naming the type associated with the identifier in the way described in "Meaning of Declarators." For example, after

```
typedef int MILES, *KLICKSP;
typedef struct { double re, im; } complex;
```

the constructions

```
MILES distance;
extern KLICKSP metricp;
complex z, *zp;
```

are all legal declarations; the type of **distance** is **int**, that of **metricp** is "pointer to **int**, " and that of **z** is the specified structure. The **zp** is a pointer to such a structure.

The **typedef** does not introduce brand-new types, only synonyms for types which could be specified in another way. Thus in the example above **distance** is considered to have exactly the same type as any other **int** object.

9. Statements

Except as indicated, statements are executed in sequence.

9.1. Expression Statement

Most statements are expression statements, which have the form

expression ;

Usually expression statements are assignments or function calls.

9.2. Compound Statement or Block

So that several statements can be used where one is expected, the compound statement (also, and equivalently, called "block") is provided:

compound-statement:
{ *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

declaration-list:
declaration
declaration declaration-list

statement-list:
statement
statement statement-list

If any of the identifiers in the declaration-list were previously declared, the outer declaration is pushed down for the duration of the block, after which it resumes its force.

Any initializations of **auto** or **register** variables are performed each time the block is entered at the top. It is currently possible (but a bad practice) to transfer into a block; in that case the initializations are not performed. Initializations of **static** variables are performed only once when the program begins execution. Inside a block, **extern** declarations do not reserve storage so initialization is not permitted.

9.3. Conditional Statement

The two forms of the conditional statement are

if (*expression*) *statement*
if (*expression*) *statement* **else** *statement*

In both cases, the expression is evaluated; and if it is nonzero, the first substatement is executed. In the second case, the second substatement is executed if the expression is 0. The "else" ambiguity is resolved by connecting an **else** with the last encountered **else**-less **if**.

9.4. While Statement

The **while** statement has the form

while (*expression*) *statement*

The substatement is executed repeatedly so long as the value of the expression remains nonzero. The test takes place before each execution of the statement.

9.5. Do Statement

The **do** statement has the form

do *statement* **while** (*expression*) ;

The substatement is executed repeatedly until the value of the expression becomes 0. The test takes place after each execution of the statement.

9.6. For Statement

The **for** statement has the form:

for (*exp-1*$_{opt}$; *exp-2*$_{opt}$; *exp-3*$_{opt}$) *statement*

Except for the behavior of **continue**, this statement is equivalent to

```
exp-1 ;
while ( exp-2 )
{
        statement
        exp-3 ;
}
```

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0. The third expression often specifies an incrementing that is performed after each iteration.

Any or all of the expressions may be dropped. A missing *exp-2* makes the implied **while** clause equivalent to **while(1)**; other missing expressions are simply dropped from the expansion above.

9.7. Switch Statement

The **switch** statement causes control to be transferred to one of several statements depending on the value of an expression. It has the form

switch (*expression*) *statement*

The usual arithmetic conversion is performed on the expression, but the result must be **int**. The statement is typically compound. Any statement within the statement may be labeled with one or more case prefixes as follows:

case *constant-expression* :

where the constant expression must be **int**. No two of the case constants in the same switch may have the same value. Constant expressions are precisely defined in "CONSTANT EXPRESSIONS."

There may also be at most one statement prefix of the form

default :

When the **switch** statement is executed, its expression is evaluated and compared with each case constant. If one of the case constants is equal to the value of the expression, control is passed to the statement following the matched case prefix. If no case constant matches the expression and if there is a **default**, prefix, control passes to the prefixed statement. If no case matches and if there is no **default**, then none of the statements in the switch is executed.

The prefixes **case** and **default** do not alter the flow of control, which continues unimpeded across such prefixes. To exit from a switch, see "Break Statement."

Usually, the statement that is the subject of a switch is compound. Declarations may appear at the head of this statement, but initializations of automatic or register variables are ineffective.

9.8. Break Statement

The statement

break ;

causes termination of the smallest enclosing **while**, **do**, **for**, or **switch** statement; control passes to the statement following the terminated statement.

9.9. Continue Statement

The statement

continue ;

causes control to pass to the loop-continuation portion of the smallest enclosing **while**, **do**, or **for** statement; that is to the end of the loop. More precisely, in each of the statements

```
while (...) {
    statement ;
    contin: ;
}
```

```
do {
    statement ;
    contin: ;
} while (...);
```

```
for (...) {
    statement ;
    contin: ;
}
```

a **continue** is equivalent to **goto contin**. (Following the **contin:** is a null statement, see "Null Statement".)

9.10. Return Statement

A function returns to its caller by means of the **return** statement which has one of the forms

return ;
return *expression* ;

In the first case, the returned value is undefined. In the second case, the value of the expression is returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value. The expression may be parenthesized.

9.11. Goto Statement

Control may be transferred unconditionally by means of the statement

goto *identifier* ;

The identifier must be a label (see "Labeled Statement") located in the current function.

9.12. Labeled Statement

Any statement may be preceded by label prefixes of the form

identifier :

which serve to declare the identifier as a label. The only use of a label is as a target of a **goto**. The scope of a label is the current function, excluding any subblocks in which the same identifier has been redeclared. See "SCOPE RULES."

9.13. Null Statement

The null statement has the form

;

A null statement is useful to carry a label just before the } of a compound statement or to supply a null body to a looping statement such as **while**.

10. External Definitions

A C program consists of a sequence of external definitions. An external definition declares an identifier to have storage class **extern** (by default) or perhaps **static**, and a specified type. The type-specifier (see "Type Specifiers" in "DECLARATIONS") may also be empty, in which case the type is taken to be **int**. The scope of external definitions persists to the end of the file in which they are declared just as the effect of declarations persists to the end of a block. The syntax of external definitions is the same as that of all declarations except that only at this level may the code for functions be given.

10.1. External Function Definitions

Function definitions have the form

function-definition:
decl-specifiers$_{opt}$ *function-declarator function-body*

The only sc-specifiers allowed among the decl-specifiers are **extern** or **static**; see "Scope of Externals" in "SCOPE RULES" for the distinction between them. A function declarator is similar to a declarator for a "function returning ..." except that it lists the formal parameters of the function being defined.

function-declarator:
declarator (parameter-list$_{opt}$ *)*

parameter-list:
identifier
identifier , parameter-list

The function-body has the form

function-body:
declaration-list$_{opt}$ *compound-statement*

The identifiers in the parameter list, and only those identifiers, may be declared in the declaration list. Any identifiers whose type is not given are taken to be **int**. The only storage class which may be specified is **register**; if it is specified, the corresponding actual parameter will be copied, if possible, into a register at the outset of the function.

A simple example of a complete function definition is

```
int max(a, b, c)
    int a, b, c;
{
    int m;

    m = (a > b) ? a : b;
    return((m > c) ? m : c);
}
```

Here **int** is the type-specifier; **max(a, b, c)** is the function-declarator; **int a, b, c;** is the declaration-list for the formal parameters; { **...** } is the block giving the code for the statement.

The C program converts all **float** actual parameters to **double**, so formal parameters declared **float** have their declaration adjusted to read **double**. All **char** and **short** formal parameter declarations are similarly adjusted to read **int**. Also, since a reference to an array in any context (in particular as an actual parameter) is taken to mean a pointer to the first element of the array, declarations of formal parameters declared "array of ..." are adjusted to read "pointer to ...."

10.2. External Data Definitions

An external data definition has the form

data-definition:
declaration

The storage class of such data may be **extern** (which is the default) or **static** but not **auto** or **register**.

11. Scope Rules

A C program need not all be compiled at the same time. The source text of the program may be kept in several files, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out both through explicit calls and through manipulation of external data.

Therefore, there are two kinds of scopes to consider: first, what may be called the lexical scope of an identifier, which is essentially the region of a program during which it may be used without drawing "undefined identifier" diagnostics; and second, the scope associated with external identifiers, which is characterized by the rule that references to the same external identifier are references to the same object.

11.1. Lexical Scope

The lexical scope of identifiers declared in external definitions persists from the definition through the end of the source file in which they appear. The lexical scope of identifiers which are formal parameters persists through the function with which they are associated. The lexical scope of identifiers declared at the head of a block persists until the end of the block. The lexical scope of labels is the whole of the function in which they appear.

In all cases, however, if an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of that identifier outside the block is suspended until the end of the block.

Remember also (see "Structure, Union, and Enumeration Declarations" in "DECLARATIONS") that tags, identifiers associated with ordinary variables, and identities associated with structure and union members form three disjoint classes which do not conflict. Members and tags follow the same scope rules as other identifiers. The **enum** constants are in the same class as ordinary variables and follow the same scope rules. The **typedef** names are in the same class as ordinary identifiers. They may be redeclared in inner blocks, but an explicit type must be given in the inner declaration:

```
typedef float distance;
...
{
    auto int distance;
    ...
}
```

The **int** must be present in the second declaration, or it would be taken to be a declaration with no declarators and type **distance**.

11.2. Scope of Externals

If a function refers to an identifier declared to be **extern**, then somewhere among the files or libraries constituting the complete program there must be at least one external definition for the identifier. All functions in a given program which refer to the same external identifier refer to the same object, so care must be taken that the type and size specified in the definition are compatible with those specified by each function which references the data.

It is illegal to explicitly initialize any external identifier more than once in the set of files and libraries comprising a multi-file program. It is legal to have more than one data definition for any external non-function identifier; explicit use of **extern** does not change the meaning of an external declaration.

In restricted environments, the use of the **extern** storage class takes on an additional meaning. In these environments, the explicit appearance of the **extern** keyword in external data declarations of identities without initialization indicates that the storage for the identifiers is allocated elsewhere, either in this file or another file. It is required that there be exactly one definition of each external identifier (without **extern**) in the set of files and libraries comprising a mult-file program.

Identifiers declared **static** at the top level in external definitions are not visible in other files. Functions may be declared **static**.

12. Compiler Control Lines

The C compiler contains a preprocessor capable of macro substitution, conditional compilation, and inclusion of named files. Lines beginning with # communicate with this preprocessor. There may be any number of blanks and horizontal tabs between the # and the directive. These lines have syntax independent of the rest of the language; they may appear anywhere and have effect which lasts (independent of scope) until the end of the source program file.

12.1. Token Replacement

A compiler-control line of the form

#define *identifier token-string*$_{opt}$

causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens. Semicolons in or at the end of the token-string are part of that string. A line of the form

#define *identifier(identifier, ...)token-string*$_{opt}$

where there is no space between the first identifier and the (, is a macro definition with arguments. There may be zero or more formal parameters. Subsequent instances of the first identifier followed by a (, a sequence of tokens delimited by commas, and a) are replaced by the token string in the definition. Each occurrence of an identifier mentioned in the formal parameter list of the definition is replaced by the corresponding token string from the call. The actual arguments in the call are token strings separated by commas; however, commas in quoted strings or protected by parentheses do not separate arguments. The number of formal and actual parameters must be the same. Strings and character constants in the token-string are scanned for formal parameters, but strings and character constants in the rest of the program are not scanned for defined identifiers to replacement.

In both forms the replacement string is rescanned for more defined identifiers. In both forms a long definition may be continued on another line by writing \ at the end of the line to be continued.

This facility is most valuable for definition of "manifest constants," as in

```
#define TABSIZE 100

int table[TABSIZE];
```

A control line of the form

#undef *identifier*

causes the identifier's preprocessor definition (if any) to be forgotten.

If a **#defined** identifier is the subject of a subsequent **#define** with no intervening **#undef**, then the two token-strings are compared textually. If the two token-strings are not identical (all white space is considered as equivalent), then the identifier is considered to be redefined.

12.2. File Inclusion

A compiler control line of the form

#include *"filename"*

causes the replacement of that line by the entire contents of the file *filename*. The named file is searched for first in the directory of the file containing the **#include**, and then in a sequence of specified or standard places. Alternatively, a control line of the form

#include *<filename>*

searches only the specified or standard places and not the directory of the **#include**. (How the places are specified is not part of the language.)

#includes may be nested.

12.3. Conditional Compilation

A compiler control line of the form

#if *restricted-constant-expression*

checks whether the restricted-constant expression evaluates to nonzero. (Constant expressions are discussed in "CONSTANT EXPRESSIONS"; the following additional restrictions apply here: the constant expression may not contain **sizeof** casts, or an enumeration constant.)

A restricted constant expression may also contain the additional unary expression

defined *identifier*

or

defined(*identifier)*

which evaluates to one if the identifier is currently defined in the preprocessor and zero if it is not.

All currently defined identifiers in restricted-constant-expressions are replaced by their token-strings (except those identifiers modified by **defined**) just as in normal text. The restricted constant expression will be evaluated only after all expressions have finished. During this evaluation, all undefined (to the procedure) identifiers evaluate to zero.

A control line of the form

#ifdef *identifier*

checks whether the identifier is currently defined in the preprocessor; i.e., whether it has been the subject of a **#define** control line. It is equivalent to **#ifdef(***identifier***)**. A control line of the form

#ifndef *identifier*

checks whether the identifier is currently undefined in the preprocessor. It is equivalent to

#if !defined(*identifier***).**

All three forms are followed by an arbitrary number of lines, possibly containing a control line

#else

and then by a control line

#endif

If the checked condition is true, then any lines between **#else** and **#endif** are ignored. If the checked condition is false, then any lines between the test and a **#else** or, lacking a **#else**, the **#endif** are ignored.

These constructions may be nested.

12.4. Line Control

For the benefit of other preprocessors which generate C programs, a line of the form

#line *constant "filename"*

causes the compiler to believe, for purposes of error diagnostics, that the line number of the next source line is given by the constant and the current input file is named by *"filename"*. If *"filename"* is absent, the remembered file name does not change.

13. Implicit Declarations

It is not always necessary to specify both the storage class and the type of identifiers in a declaration. The storage class is supplied by the context in external definitions and in declarations of formal parameters and structure members. In a declaration inside a function, if a storage class but no type is given, the identifier is assumed to be **int**; if a type but no storage class is indicated, the identifier is assumed to be **auto**. An exception to the latter rule is made for functions because **auto** functions do not exist. If the type of an identifier is "function returning ...," it is implicitly declared to be **extern**.

In an expression, an identifier followed by (and not already declared is contextually declared to be "function returning **int**."

14. Types Revisited

This part summarizes the operations which can be performed on objects of certain types.

14.1. Structures and Unions

Structures and unions may be assigned, passed as arguments to functions, and returned by functions. Other plausible operators, such as equality comparison and structure casts, are not implemented.

In a reference to a structure or union member, the name on the right of the -> or the . must specify a member of the aggregate named or pointed to by the expression on the left. In general, a member of a union may not be inspected unless the value of the union has been assigned using that same member. However, one special guarantee is made by the language in order to simplify the use of unions: if a union contains several structures that share a common initial sequence and if the union currently contains one of these structures, it is permitted to inspect the common initial part of any of the contained structures. For example, the following is a legal fragment:

```
union
{
        struct
        {
                int     type;
        } n;
        struct
        {
                int     type;
                int     intnode;
        } ni;
        struct
        {
                int     type;
                float   floatnode;
        } nf;
} u;
...
u.nf.type = FLOAT;
u.nf.floatnode = 3.14;
...
if (u.n.type == FLOAT)
        ... sin(u.nf.floatnode) ...
```

14.2. Functions

There are only two things that can be done with a function **m**, call it or take its address. If the name of a function appears in an expression not in the function-name position of a call, a pointer to the function is generated. Thus, to pass one function to another, one might say

```
int f();
...
g(f);
```

Then the definition of **g** might read

```
g(funcp)
        int (*funcp)();
{
        ...
        (*funcp)();
        ...
}
```

Notice that **f** must be declared explicitly in the calling routine since its appearance in **g(f)** was not followed by (.

14.3. Arrays, Pointers, and Subscripting

Every time an identifier of array type appears in an expression, it is converted into a pointer to the first member of the array. Because of this conversion, arrays are not lvalues. By definition, the subscript operator **[]** is interpreted in such a way that **E1[E2]** is identical to ***((E1)+E2))**. Because of the conversion rules which apply to +, if **E1** is an array and **E2** an integer, then **E1[E2]** refers to the **E2-th** member of **E1**. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

A consistent rule is followed in the case of multidimensional arrays. If E is an *n*-dimensional array of rank i×j×...×k, then E appearing in an expression is converted to a pointer to an (n-1)-dimensional array with rank j×...×k. If the * operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to (n-1)-dimensional array, which itself is immediately converted into a pointer.

For example, consider

```
int x[3][5];
```

Here **x** is a 3×5 array of integers. When **x** appears in an expression, it is converted to a pointer to (the first of three) 5-membered arrays of integers. In the expression **x[i]**, which is equivalent to ***(x+i)**, **x** is first converted to a pointer as described; then **i** is converted to the type of **x**, which involves multiplying **i** by the length the object to which the pointer points, namely 5-integer objects. The results are added and indirection applied to yield an array (of five integers) which in turn is converted to a pointer to the first of the integers. If there is another subscript, the same argument applies again; this time the result is an integer.

Arrays in C are stored row-wise (last subscript varies fastest) and the first subscript in the declaration helps determine the amount of storage consumed by an array. Arrays play no other part in subscript calculations.

14.4. Explicit Pointer Conversions

Certain conversions involving pointers are permitted but have implementation-dependent aspects. They are all specified by means of an explicit type-conversion operator, see "Unary Operators" under"EXPRESSIONS" and "Type Names"under "DECLARATIONS."

A pointer may be converted to any of the integral types large enough to hold it. Whether an **int** or **long** is required is machine dependent. The mapping function is also machine dependent but is intended to be unsurprising to those who know the addressing structure of the machine. Details for some particular machines are given below.

An object of integral type may be explicitly converted to a pointer. The mapping always carries an integer converted from a pointer back to the same pointer but is otherwise machine dependent.

A pointer to one type may be converted to a pointer to another type. The resulting pointer may cause addressing exceptions upon use if the subject pointer does not refer to an object suitably aligned in storage. It is guaranteed that a pointer to an object of a given size may be converted to a pointer to an object of a smaller size and back again without change.

For example, a storage-allocation routine might accept a size (in bytes) of an object to allocate, and return a **char** pointer; it might be used in this way.

```
extern char *malloc();
double *dp;

dp = (double *) malloc(sizeof(double));
*dp = 22.0 / 7.0;
```

The **alloc** must ensure (in a machine-dependent way) that its return value is suitable for conversion to a pointer to **double**; then the *use* of the function is portable.

The pointer representation on the PDP-11 corresponds to a 16-bit integer and measures bytes. The **char**'s have no alignment requirements; everything else must have an even address.

On the VAX-11, pointers are 32 bits long and measure bytes. Elementary objects are aligned on a boundary equal to their length, except that **double** quantities need be aligned only on even 4-byte boundaries. Aggregates are aligned on the strictest boundary required by any of their constituents.

The 3B 20 computer has 24-bit pointers placed into 32-bit quantities. Most objects are aligned on 4-byte boundaries. **Shorts** are aligned in all cases on 2-byte boundaries. Arrays of characters, all structures, **int**s, **long**s, **float**s, and **double**s are aligned on 4-byte boundries; but structure members may

be packed tighter.

14.5. CONSTANT EXPRESSIONS

In several places C requires expressions that evaluate to a constant: after **case**, as array bounds, and in initializers. In the first two cases, the expression can involve only integer constants, character constants, casts to integral types, enumeration constants, and **sizeof** expressions, possibly connected by the binary operators

```
+ - * / % & | ^ << >> == != < > <= >= && ||
```

or by the unary operators

```
- ~
```

or by the ternary operator

```
?:
```

Parentheses can be used for grouping but not for function calls.

More latitude is permitted for initializers; besides constant expressions as discussed above, one can also use floating constants and arbitrary casts and can also apply the unary & operator to external or static objects and to external or static arrays subscripted with a constant expression. The unary & can also be applied implicitly by appearance of unsubscripted arrays and functions. The basic rule is that initializers must evaluate either to a constant or to the address of a previously declared external or static object plus or minus a constant.

15. Portability Considerations

Certain parts of C are inherently machine dependent. The following list of potential trouble spots is not meant to be all-inclusive but to point out the main ones.

Purely hardware issues like word size and the properties of floating point arithmetic and integer division have proven in practice to be not much of a problem. Other facets of the hardware are reflected in differing implementations. Some of these, particularly sign extension (converting a negative character into a negative integer) and the order in which bytes are placed in a word, are nuisances that must be carefully watched. Most of the others are only minor problems.

The number of **register** variables that can actually be placed in registers varies from machine to machine as does the set of valid types. Nonetheless, the compilers all do things properly for their own machine; excess or invalid **register** declarations are ignored.

Some difficulties arise only when dubious coding practices are used. It is exceedingly unwise to write programs that depend on any of these properties.

The order of evaluation of function arguments is not specified by the language. The order in which side effects take place is also unspecified.

Since character constants are really objects of type **int**, multicharacter character constants may be permitted. The specific implementation is very machine dependent because the order in which characters are assigned to a word varies from one machine to another.

Fields are assigned to words and characters to integers right to left on some machines and left to right on other machines. These differences are invisible to isolated programs that do not indulge in type punning (e.g., by converting an **int** pointer to a **char** pointer and inspecting the pointed-to storage) but must be accounted for when conforming to externally-imposed storage layouts.

16. Syntax Summary

This summary of C syntax is intended more for aiding comprehension than as an exact state-

ment of the language.

16.1. Expressions

The basic expressions are:

expression:
primary
* *expression*
&*lvalue*
\- *expression*
! expression
~ *expression*
++ *lvalue*
--*lvalue*
lvalue ++
lvalue --
sizeof *expression*
sizeof (*type-name***)**
(type-name) expression
expression binop expression
expression ? expression : expression
lvalue asgnop expression
expression , expression

primary:
identifier
constant
string
(expression)
primary (expression-list$_{opt}$ *)*
primary [expression]
primary . identifier
primary – identifier

lvalue:
identifier
primary [expression]
lvalue . identifier
primary – identifier
* *expression*
(lvalue)

The primary-expression operators

() [] . –

have highest priority and group left to right. The unary operators

* & – ! ~ ++ -- **sizeof** *(type-name)*

have priority below the primary operators but higher than any binary operator and group right to left. Binary operators group left to right; they have priority decreasing as indicated below.

binop:

```
*   /   %
+   -
>>  <<
<   >   <=   >=
==  !=
&
^
|
&&
||
```

The conditional operator groups right to left.

Assignment operators all have the same priority and all group right to left.

asgnop:

```
=  +=  -=  *=  /=  %=  >>=  <<=  &=  ^=  |=
```

The comma operator has the lowest priority and groups left to right.

16.2. Declarations

declaration:
decl-specifiers init-declarator-list$_{opt}$;

decl-specifiers:
type-specifier decl-specifiers$_{opt}$
sc-specifier decl-specifiers$_{opt}$

sc-specifier:
auto
static
extern
register
typedef

type-specifier:
struct-or-union-specifier
typedef-name
enum-specifier

basic-type-specifier:
basic-type
basic-type basic-type-specifiers

basic-type:
char
short
int
long
unsigned
float
double
void

enum-specifier:
enum { *enum-list* }
enum *identifier* { *enum-list* }
enum *identifier*

enum-list:
enumerator
enum-list , *enumerator*

enumerator:
identifier
identifier = *constant-expression*

init-declarator-list:
init-declarator
init-declarator , *init-declarator-list*

init-declarator:
declarator initializer$_{opt}$

declarator:
identifier
(declarator)
* *declarator*
declarator ()
declarator [constant-expression$_{opt}$ *]*

struct-or-union-specifier:
struct { *struct-decl-list* }
struct *identifier* { *struct-decl-list* }
struct *identifier*
union { *struct-decl-list* }
union *identifier* { *struct-decl-list* }
union *identifier*

struct-decl-list:
struct-declaration
struct-declaration struct-decl-list

struct-declaration:
type-specifier struct-declarator-list ;

struct-declarator-list:
struct-declarator
struct-declarator , *struct-declarator-list*

struct-declarator:
declarator
declarator : constant-expression
: constant-expression

initializer:
= *expression*
= { *initializer-list* }
= { *initializer-list* , }

initializer-list:
expression
initializer-list , *initializer-list*
{ *initializer-list* }
{ *initializer-list* , }

type-name:
type-specifier abstract-declarator

abstract-declarator:
empty
(abstract-declarator)
* *abstract-declarator*
abstract-declarator ()
abstract-declarator [constant-expression$_{opt}$ *]*

typedef-name:
identifier

16.3. Statements

compound-statement:
{ *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

declaration-list:
declaration
declaration declaration-list

statement-list:
statement
statement statement-list

statement:
compound-statement
expression ;
if *(expression) statement*
if *(expression) statement* **else** *statement*
while *(expression) statement*
do *statement* **while** *(expression) ;*
for *(exp_{opt};exp_{opt};exp_{opt}) statement*
switch *(expression) statement*
case *constant-expression : statement*
default *: statement*
break ;
continue ;
return ;
return *expression ;*
goto *identifier ;*
identifier : statement
;

16.4. External definitions

program:
external-definition
external-definition program

external-definition:
function-definition
data-definition

function-definition:
decl-specifier$_{opt}$ *function-declarator function-body*

function-declarator:
declarator (parameter-list$_{opt}$ *)*

parameter-list:
identifier
identifier , parameter-list

function-body:
declaration-list$_{opt}$ *compound-statement*

data-definition:
extern *declaration ;*
static *declaration ;*

17. Preprocessor

#define *identifier token-string*$_{opt}$
#define *identifier*(*identifier*,...)*token-string*$_{opt}$
#undef *identifier*
#include "*filename*"
#include <*filename*>
#if *restricted-constant-expression*
#ifdef *identifier*
#ifndef *identifier*
#else
#endif
#line *constant* "*filename*"

A Portable Fortran 77 Compiler

S. I. Feldman

P. J. Weinberger

Bell Laboratories
Murray Hill, New Jersey 07974

J. Berkman

University of California
Berkeley, CA 94720

ABSTRACT

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

This is a standard Bell Laboratories document reproduced with minor modifications to the text. The Bell Laboratory's appendix on "Differences Between Fortran 66 and Fortran 77" has been changed to Appendix A, and a local appendix has been added. Appendix B contains a list of Fortran 77 references (some from the original Bell document and some added at Berkeley).

Revised September, 1985

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

Table of Contents

1. INTRODUCTION

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

1.1. Usage

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

f77 *flags file* . . .

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

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

Arguments whose names end with **.f** are taken to be Fortran 77 source programs; they are compiled, and each object program is left on the file in the current directory whose name is that of the source with **.o** substituted for **.f**.

Arguments whose names end with **.F** are also taken to be Fortran 77 source programs; these are first processed by the C preprocessor before being compiled by **f77**.

Arguments whose names end with **.r** or **.e** are taken to be Ratfor or EFL source programs, respectively; these are first transformed by the appropriate preprocessor, then compiled by **f77**.

In the same way, arguments whose names end with **.c** or **.s** are taken to be C or assembly source programs and are compiled or assembled, producing a **.o** file.

The following flags are understood:

-c	Compile but do not load. Output for **x.f**, **x.F**, **x.e**, **x.r**, **x.c**, or **x.s** is put on file **x.o**.
-d	Used in debugging the compiler.
-g	Have the compiler produce additional symbol table information for *dbx(1)*. This flag is incompatible with **-O**. See section 1.4 for more details.
-i2	On machines which support short integers, make the default integer constants and variables short (see section 2.14). (**-i4** is the standard value of this option). All log-

ical quantities will be short.

-m Apply the M4 macro preprocessor to each EFL or Ratfor source file before using the appropriate compiler.

-o *file* Put executable module on file *file*. (Default is **a.out**).

-onetrip or **-1**
Compile code that performs every **do** loop at least once (see section 2.12).

-p Generate code to produce usage profiles.

-pg Generate code in the manner of **-p**, but invoke a run-time recording mechanism that keeps more extensive statistics. See *gprof*(1).

-q Suppress printing of file names and program unit names during compilation.

-r8 Treat all floating point variables, constants, functions and intrinsics as double precision and all complex quantities as double complex. See section 2.17.

-u Make the default type of a variable **undefined** (see section 2.3).

-v Print the version number of the compiler and the name of each pass.

-w Suppress all warning messages.

-w66 Suppress warnings about Fortran 66 features used.

-C Compile code that checks that subscripts are within array bounds. For multi-dimensional arrays, only the equivalent linear subscript is checked.

-D*name=def*

-D*name* Define the *name* to the C preprocessor, as if by '#define'. If no definition is given, the name is defined as "1". (**.F** files only).

-E*str* Use the string *str* as an EFL option in processing **.e** files.

-F Ratfor, EFL, and **.F** source files are pre-processed into **.f** files, and those **.f** files are left on the disk without being compiled.

-I*dir* '#include' files whose names do not begin with '/' are always sought first in the directory of the *file* argument, then in directories named in **-I** options, then in directories on a standard list. (**.F** files only).

-N[qxscn]*nnn*
Make static tables in the compiler bigger. The compiler will complain if it overflows its tables and suggest you apply one or more of these flags. These flags have the following meanings:

- **q** Maximum number of equivalenced variables. Default is 150.
- **x** Maximum number of external names (common block names, subroutine and function names). Default is 200.
- **s** Maximum number of statement numbers. Default is 401.
- **c** Maximum depth of nesting for control statements (e.g. DO loops). Default is 20.
- **n** Maximum number of identifiers. Default is 1009.

-O Invoke the object code optimizer. Incompatible with **-g**.

-R*str* Use the string *str* as a Ratfor option in processing **.r** files.

-U Do not convert upper case letters to lower case. The default is to convert Fortran programs to lower case except within character string constants.

-S Generate assembler output for each source file, but do not assemble it. Assembler output for a source file **x.f**, **x.F**, **x.e**, **x.r**, or **x.c** is put on file **x.s**.

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

1.2. Documentation Conventions

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

1.3. Implementation Strategy

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

1.4. Debugging Aids

A memory image is sometimes written to a file **core** in the current directory upon abnormal termination for errors caught by the **f77** libraries, user calls to **abort**, and certain signals (see **sigvec**(2) in the *UNIX Programmer's Manual*). **Core** is normally created only if the **-g** flag was specified to **f77** during loading.† The source-level debugger *dbx*(1) may be used with the executable and the **core** file to examine the image and determine what went wrong.

In the event that it is necessary to override this default behavior, the user may set the environment variable **f77_dump_flag**. If **f77_dump_flag** is set to a value beginning with **n**, a **core** file is not produced regardless of whether **-g** was specified at compile time, and if the value begins with **y**, dumps are produced even if **-g** was not specified.

2. LANGUAGE EXTENSIONS

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

In addition to implementing the language specified in the new Standard, our compiler implements a few extensions described in this section. Most are useful additions to the language. The remainder are extensions to make it easier to communicate with C procedures or to permit compilation of old (1966 Standard) programs.

2.1. Double Complex Data Type

The new type **double complex** is defined. Each datum is represented by a pair of double precision real values. The statements

```
z1 = ( 0.1d0, 0.2d0 )
z2 = dcmplx( dx, dy )
```

assign double complex values to **z1** and **z2**. The double precision values which constitute the double complex value may be isolated by using **dreal** or **dble** for the real part and **imag** or **dimag** for the imaginary part. To compute the double complex conjugate of a double complex value, use **conjg** or **dconjg**. The other **double complex** intrinsic functions may be accessed using their generic names or specific names. The generic names are: **abs**, **sqrt**, **exp**, **log**, **sin**, and **cos**. The specific names are the same as the generic names preceded by either **cd** or **z**, e.g. you may code **sqrt**, **zsqrt** or **cdsqrt** to compute the square root of a double complex value.

†Specify **-g** when loading with **cc** or **f77**; specify **-lg** as a library when using **ld** directly.

2.2. Internal Files

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

2.3. Implicit Undefined Statement

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

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

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

2.4. Recursion

Procedures may call themselves, directly or through a chain of other procedures. Since Fortran variables are by default **static**, it is often necessary to use the **automatic** storage extension to prevent unexpected results from recursive functions.

2.5. Automatic Storage

Two new keywords are recognized, **static** and **automatic.** These keywords may appear as "types" in type statements and in **implicit** statements. Local variables are static by default; there is only one instance of the variable. For variables declared **automatic,** there is a separate instance of the variable for each invocation of the procedure. Automatic variables may not appear in **equivalence, data,** or **save** statements. Neither type of variable is guaranteed to retain its value between calls to a subprogram (see the **save** statement in Appendix A).

2.6. Source Input Format

The Standard expects input to the compiler to be in 72-column format: except in comment lines, the first five characters are the statement number, the next is the continuation character, and the next 66 are the body of the line. (If there are fewer than 72 characters on a line, the compiler pads it with blanks; characters after the seventy-second are ignored.)

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

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

2.7. Include Statement

The statement

```
include 'stuff'
```

is replaced by the contents of the file **stuff**; **include** statements may be nested to a reasonable

depth, currently ten.

2.8. Binary Initialization Constants

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

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

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

2.9. Character Strings

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

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

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

Each character string constant appearing outside a **data** statement is followed by a null character to ease communication with C routines.

2.10. Hollerith

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

2.11. Equivalence Statements

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

2.12. One-Trip DO Loops

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

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

The 1966 Standard stated that the effect of such a statement was undefined, but it was common practice that the range of a **do** loop would be performed at least once. In order to accommodate old programs, though they were in violation of the 1966 Standard, the **-onetrip** or **-1** compiler

flags causes non-standard loops to be generated.

2.13. Commas in Formatted Input

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

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

will read the record

```
-345,.05e-3,12
```

correctly.

2.14. Short Integers

On machines that support halfword integers, the compiler accepts declarations of type **integer*2.** (Ordinary integers follow the Fortran rules about occupying the same space as a real variable; they are assumed to be of C type **long int**; halfword integers are of C type **short int.**) An expression involving only objects of type **integer*2** is of that type. Generic functions return short or long integers depending on the actual types of their arguments. If a procedure is compiled using the **-i2** flag, all small integer constants will be of type **integer*2.** If the precision of an integer-valued intrinsic function is not determined by the generic function rules, one will be chosen that returns the prevailing length (**integer*2** when the **-i2** command flag is in effect). When the **-i2** option is in effect, all quantities of type **logical** will be short. Note that these short integer and logical quantities do not obey the standard rules for storage association.

2.15. Additional Intrinsic Functions

This compiler supports all of the intrinsic functions specified in the Fortran 77 Standard. In addition, there are built-in functions for performing bitwise logical and boolean operations on integer and logical values (**or, and, xor, not, lshift,** and **rshift**), and intrinsic functions for **double complex** values (see section 2.1). The **f77** library contains many other functions, such as accessing the UNIX command arguments (**getarg** and **iargc**) and environment (**getenv**). See **intro**(3f) and **bit**(3f) in the *UNIX Programmer's Manual* for more information.

2.16. Namelist I/O

Namelist I/O provides an easy way to input and output information without formats. Although not part of the standard, namelist I/O was part of many Fortran 66 systems and is a common extension to Fortran 77 systems.

Variables and arrays to be used in namelist I/O are declared as part of a namelist in a **namelist** statement, e.g.:

```
character str*12
logical flags(20)
complex c(2)
real arr1(2,3), arr2(0:3,4)
namelist /basic/ arr1, arr2, key, str, c /flglst/ key, flags
```

This defines two namelists: list **basic** consists of variables **key** and **str** and arrays **arr1, arr2,** and **c**; list **flglst** consists of variable **key** and array **flags.** A namelist can include variables and arrays of any type, and a variable or array may be in several different namelists. However dummy arguments and array elements may not be in a namelist. A namelist name may be used in external sequential **read, write** and **print** statements wherever a format could be used.

In a namelist **read**, column one of each data record is ignored. The data begins with an ampersand in column 2 followed by the namelist name and a blank. Then there is a sequence of value assignments separated by commas and finally an "&end". A simple example of input data

corresponding to namelist **basic** is:

```
&basic key=5, str='hi there' &end
```

For compatibility with other systems, dollar signs may be used instead of the ampersands:

```
$basic key=5, str='hi there' $end
```

A value assignment in the data record must be one of three forms. The simplest is a variable name followed by an equal sign followed by a data value which is assigned to that variable, e.g. "key=5". The second form consists of an array name followed by "=" followed by one or more values to be assigned to the array, e.g.:

```
c=(1.1,-2.9),(-1.8e+10,14.0e-3)
```

assigns values to c(1) and c(2) in the complex array c.

As in other **read** statements, values are assigned in the order of the array in memory, i.e. column-major order for two dimensional arrays. Multiple copies of a value may be represented by a repetition count followed by an asterisk followed by the value; e.g. "3*55.4" is the same as "55.4, 55.4, 55.4". It is an error to specify more values than the array can hold; if less are specified, only that number of elements of the array are changed. The third form of a value assignment is a subscripted variable name followed by "=" followed by a value or values, e.g.: "arr2(0,4)=15.2". Only integer constant subscripts may be used. The correct number of subscripts must be used and the subscripts must be legal. This form is the same as the form with an array name except the array is filled starting at the named element.

In all three forms, the variable or array name must be declared in the namelist. The form of the data values is the same as in list directed input except that in namelist I/O, character strings in the data must be enclosed in apostrophes or double quotes, and repetition counts must be followed by data values.

One use of namelist input is to read in a list of options or flags. For example:

```
      logical flags(14)
      namelist /pars/ flags, iters, xlow, xhigh, xinc
      data flags/14*.false./

10    read(5,pars,end=900)
      print pars
      call calc( xlow, xhigh, xinc, flags, iters )
      go to 10
900   continue
      end
```

could be run with the following data (each record begins with a space):

```
&pars iters=10, xlow=0.0, xhigh=1.0, xinc=0.1 &end
&pars xinc=0.2,
  flags(2)=2*.true., flags(8)=.true. &end
&pars xlow=2.0, xhigh=8.0 &end
```

The program reads parameters for the run from the first data set and computes using them. Then it loops and each successive set of namelist input data specifies only those data items which need to be changed. Note the second data set sets the 2^{nd}, 3^{rd}, and 8^{th} elements in the array **flags** to **.true.**.

When a namelist name is used in a **write** or **print** statement, all the values in the namelist are output together with their names. For example the **print** in the program above prints the following:

```
&pars flags= f, f, f, f, f, f, f, f, f, f, f, f, f, f, iters=
 10, xlow= 0., xhigh= 1.00000, xinc= 0.100000
&end
&pars flags= f, t, t, f, f, f, f, t, f, f, f, f, f, f, iters=
 10, xlow= 0., xhigh= 1.00000, xinc= 0.200000
&end
&pars flags= f, t, t, f, f, f, f, t, f, f, f, f, f, f, iters=
 10, xlow= 2.00000, xhigh= 8.00000, xinc= 0.200000
&end
```

Each line begins with a space so that namelist output can be used as input to a namelist **read**. The default is to use ampersands in namelist **print** and **write**. However, dollar signs will be used if the last preceding namelist **read** data set used dollar signs. The character to be used is stored as the first character of the common block **namelistkey**.

2.17. Automatic Precision Increase

The **-r8** flag allows a user to run a program with increased precision without changing any of the program source, i.e. it allows a user to take a program coded in single precision and compile and execute it as if it had been coded in double precision. The option extends the precision of all single precision real and complex constants, variables, external functions, and intrinsic functions. For example, the source:

```
implicit complex(c)
real last
intrinsic sin, csin
data last/0.3/

x = 0.1
y = sqrt(x)+sqrt(last)
c1 = (0.1,0.2)
c2 = sqrt(c1)
x = real(i)
y = aimag(c1)
call fun(sin,csin)
```

is compiled under this flag as if it had been written as:

```
implicit double precision (a-b,d-h,o-z), double complex(c)
double precision last
intrinsic dsin, cdsin
data last/0.3d0/

x = 0.1d0
y = sqrt(x)+sqrt(last)
c1 = (0.1d0,0.2d0)
c2 = sqrt(c1)
x = dreal(i)
y = dimag(c1)
call fun(dsin,cdsin)
```

When the **-r8** flag is invoked, the calls using the generic name **sqrt** will refer to a different specific function since the types of the arguments have changed. This option extends the precision of all single precision **real** and **complex** variables and functions, including those declared **real*4** and **complex*8**.

In order to successfully use this flag to increase precision, the entire program including all the subroutines and functions it calls must be recompiled. Programs which use dynamic memory

allocation or use equivalence or common statements to associate variables of different types may have to be changed by hand. Similar caveats apply to the sizes of records in unformatted I/O.

2.18. Characters and Integers

A character constant of integer length or less may be assigned to an integer variable. Individual bytes are packed into the integer in the native byte order. The character constant is padded with blanks to the width of the integer during the assignment. Use of this feature is deprecated; it is intended only as a porting aid for extended Fortran 66 programs. Note that the intrinsic **ichar** function behaves as the standard requires, converting only single bytes to integers.

3. VIOLATIONS OF THE STANDARD

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

3.1. Double Precision Alignment

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

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

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

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

3.2. Dummy Procedure Arguments

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

3.3. T and TL Formats

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

3.4. Carriage Control

The Standard leaves as implementation dependent which logical unit(s) are treated as "printer" files. In this implementation there is no printer file and thus by default, no carriage control is recognized on formatted output. This can be changed using **form='print'** in the **open** statement for a unit, or by using the **fpr**(1) filter for output; see [9].

3.5. Assigned Goto

The optional *list* associated with an assigned **goto** statement is not checked against the actual assigned value during execution.

4. INTER-PROCEDURE INTERFACE

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

4.1. Procedure Names

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

4.2. Data Representations

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

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

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

4.3. Arrays

The first element of a C array always has subscript zero, while Fortran arrays begin at 1 by default. Fortran arrays are stored in column-major order in contiguous storage, C arrays are stored in row-major order. Many mathematical libraries have subroutines which transpose a two dimensional matrix, e.g. **f01crf** in the **NAG** library and **vtran** in the **IMSL** library. These may be used to transpose a two-dimensional array stored in C in row-major order to Fortran column-major order or vice-versa.

4.4. Return Values

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

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

is equivalent to

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

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

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

is equivalent to

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

and could be invoked in C by

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

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

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

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

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

4.5. Argument Lists

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

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

Thus, the call in

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

is equivalent to that in

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

4.6. System Interface

To run a Fortran program, the system invokes a small C program which first initializes signal handling, then calls **f_init** to initialize the Fortran I/O library, then calls your Fortran main program, and then calls **f_exit** to close any Fortran files opened.

f_init initializes Fortran units 0, 5, and 6 to standard error, standard input, and standard output respectively. It also calls **setlinebuf** to initiate line buffering of standard error. If you are using Fortran subroutines which may do I/O and you have a C main program, call **f_init** before calling the Fortran subroutines. Otherwise, Fortran units 0, 5, and 6 will be connected to files **fort.0**, **fort.5**, and **fort.6**, and error messages from the **f77** libraries will be written to **fort.0** instead of to standard error. If your C program terminates by calling the C function **exit**, all files are

automatically closed. If there are Fortran scratch files to be deleted, first call **f_exit**. **F_init** and **f_exit** do not have any arguments.

The **-d** flag will show what libraries are used in loading Fortran programs.

5. FILE FORMATS

5.1. Structure of Fortran Files

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

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

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

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

5.2. Portability Considerations

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

5.3. Logical Units and Files

Fortran logical unit numbers may be any integer between 0 and 99. The number of simultaneously open files is currently limited to 48.

Units 5, 6, and 0 are connected before the program begins to standard input, standard output, and standard error respectively.

If an unit is opened explicitly by an **open** statement with a **file=** keyword, then the file name is the name from the **open** statement. Otherwise, the default file name corresponding to unit *n* is **fort.***n*. If there is an environment variable whose name is the same as the tail of the file name after periods are deleted, then the contents of that environment variable are used as the name of the file. See [9] for details.

The default connection for all units is for sequential formatted I/O. The Standard does not specify where a file which has been explicitly **open**ed for sequential I/O is initially positioned. The I/O system will position the file at the beginning. Therefore a **write** will destroy any data

already in the file, but a **read** will work reasonably. To position a file to its end, use a **read** loop, or the system dependent function **fseek**. The preconnected units 0, 5, and 6 are positioned as they come from the program's parent process.

APPENDIX A: Differences Between Fortran 66 and Fortran 77

The following is a very brief description of the differences between the 1966 [2] and the 1977 [1] Standard languages. We assume that the reader is familiar with Fortran 66. We do not pretend to be complete, precise, or unbiased, but plan to describe what we feel are the most important aspects of the new language. The best current information on the 1977 Standard is in publications of the X3J3 Subcommittee of the American National Standards Institute, and the ANSI X3.9-1978 document, the official description of the language. The Standard is written in English rather than a meta-language, but it is forbidding and legalistic. A number of tutorials and textbooks are available (see Appendix B).

1. Features Deleted from Fortran 66

1.1. Hollerith

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

1.2. Extended Range of DO

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

2. Program Form

2.1. Blank Lines

Completely blank lines are now legal comment lines.

2.2. Program and Block Data Statements

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

```
program work
```

Block data procedures may also have names.

```
block data stuff
```

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

2.3. ENTRY Statement

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

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

Execution begins at the first statement following the **entry** line. All variable declarations must precede all executable statements in the procedure. If the procedure begins with a **subroutine** statement, all entry points are subroutine names. If it begins with a **function** statement, each entry is a function entry point, with type determined by the type declared for the entry name. If any entry is a character-valued function, then all entries must be. In a function, an entry name of the same type as that where control entered must be assigned a value. Arguments do not retain their values between calls. (The ancient trick of calling one entry point with a large number of arguments to cause the procedure to "remember" the locations of those arguments, then invoking an entry with just a few arguments for later calculation, is still illegal.

Furthermore, the trick doesn't work in our implementation, since arguments are not kept in static storage.)

2.4. DO Loops

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

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

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

2.5. Alternate Returns

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

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

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

3. Declarations

3.1. CHARACTER Data Type

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

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

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

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

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

3.2. IMPLICIT Statement

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

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

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

3.3. PARAMETER Statement

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

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

The type of each parameter name is governed by the same implicit and explicit rules as for a variable. Symbolic names for **character** constants may be declared with an implied length "(*)". The right side of each equal sign must be a constant expression (an expression made up of constants, operators, and already defined parameters).

3.4. Array Declarations

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

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

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

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

3.5. SAVE Statement

A little known rule of Fortran 66 is that variables in a procedure do not necessarily retain their values between invocations of that procedure. This rule permits overlay and stack implementations for the affected variables. In Fortran 77, three types of variables automatically keep there values: variables in blank common, variables defined in **data** statements and never changed, and variables in named common blocks which have not become undefined. At any instant in the execution of a program, if a named common block is declared neither in the currently executing procedure nor in any of the procedures in the chain of callers, all of the variables in that common block become undefined. Fortran 77 permits one to specify that certain variables and common blocks are to retain their values between invocations. The declaration

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

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

```
save
```

has this effect on all variables and common blocks in the procedure. A common block must be **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'
```

Although null (zero-length) character strings are not allowed in the standard Fortran, they may be used with f77. Our compiler has two different quotation marks, " ' " and " " ". (See section 2.9 in the main text.)

4.2. Concatenation

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

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

are equal.

Dummy arguments of type character may be declared with implied lengths:

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

Such dummy arguments may be used in concatenations in assign statements:

```
s = a // b
```

but not in other contexts. For example:

```
if( a // b .eq. 'abc' ) key = 1
call sub( a // b )
```

are legal statements if "a" and "b" are dummy arguments declared with explicit lengths, or if they are not arguments. These are illegal if they are declared with implied lengths.

4.3. Character String Assignment

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

```
str = ' ' // str
str = str(2:)
```

These are not flagged as errors during compilation or execution, however the result is undefined.

4.4. Substrings

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

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

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

4.5. Exponentiation

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

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

4.6. Relaxation of Restrictions

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

Constant expressions are permitted where a constant is allowed, except in **data** statements and **format** statements. (A constant expression is made up of explicit constants and **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 **common.**

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

5. Executable Statements

5.1. IF-THEN-ELSE

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

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

and ends with an

```
end if
```

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

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

statements, followed by at most one

```
else
```

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

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

5.2. Alternate Returns

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

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

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

```
return k
```

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

statement following the **call** is executed.

6. Input/Output

6.1. Format Variables

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

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

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

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

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

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

6.3. Formatted I/O

6.3.1. Character Constants

Character constants in formats are copied literally to the output.

A format may be specified as a character constant within the **read** or **write** statement.

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

produces

```
 7 isn't 4
```

In the example above, the format is the character constant

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

and the embedded character constant

```
 isn't
```

is copied into the output.

The example could have been written more legibly by taking advantage of the two types of quote marks.

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

However, the double quote is not standard Fortran 77.

The standard does not allow reading into character constants or Hollerith fields. In order to facilitate running older programs, the Fortran I/O library allows reading into Hollerith fields; however this is a practice to be avoided.

6.3.2. Positional Editing Codes

t, tl, tr, and **x** codes control where the next character is in the record. **tr***n* or *n***x** specifies that the next character is *n* to the right of the current position. **tl***n* specifies that the next character is *n* to the left of the current position, allowing parts of the record to be reconsidered. **t***n* says that the next character is to be character number *n* in the record. (See section 3.3 in the main text.)

6.3.3. Colon

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

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

the first **write** statement prints

```
hello there 12
```

while the second only prints

```
hello
```

6.3.4. Optional Plus Signs

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

6.3.5. Blanks on Input

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

6.3.6. Unrepresentable Values

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

6.3.7. Iw.m

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

6.3.8. Floating Point

On input, exponents may start with the letter **E, D, e,** or **d**. All have the same meaning. On output we always use **e** or **d**. The **e** and **d** format codes also have identical meanings. A leading zero before the decimal point in **e** output without a scale factor is optional with the implementation. There is a **g***w.d* format code which is the same as **e***w.d* and **f***w.d* on input, but which chooses **f** or **e** formats for output depending on the size of the number and of *d*.

PS 2 1

6.3.9. "A" Format Code

The **a** code is used for character data. **a***w* uses a field width of *w*, while a plain **a** uses the length of the internal character item.

6.4. Standard Units

There are default formatted input and output units. The statement

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

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

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

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

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

6.5. List-Directed I/O

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

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

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

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

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

The Fortran standard requires data being read into **character** variables by a list-directed read to be enclosed in quotes. In our system, the quotes are optional for strings which do not start with a digit or quote and do not contain separators.

For output, suitable formats are chosen for each item. The values of character strings are printed; they are not enclosed in quotes. According to the standard, they could not be read back using list-directed input. However much of this data could be read back in with list-directed I/O on our system.

6.6. Direct I/O

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

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

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

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

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

6.7. Internal Files

Internal files are character string objects, such as variables or substrings, or arrays of type character. In the former cases there is only a single record in the file; in the latter case each array element is a record. The Standard includes only sequential formatted I/O on internal files. (I/O is not a very precise term to use here, but internal files are dealt with using **read** and **write.**) Internal files are used by giving the name of the character object in place of the unit number, as in

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

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

We also support two extensions of the standard. The first is direct I/O on internal files. This is like direct I/O on external files, except that the number of records in the file cannot be changed. In this case a record is a single element of an array of character strings. The second extension is list-directed I/O on internal files.

6.8. OPEN, CLOSE, and INQUIRE Statements

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

6.8.1. OPEN

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

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

open takes a variety of arguments with meanings described below.

unit= an integer between 0 and 99 inclusive which is the unit to which the file is to be connected (see section 5.3 in the text). If this parameter is the first one in the **open** statement, the **unit=** can be omitted.

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

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

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

status= one of **'old', 'new', 'scratch',** or **'unknown'.** If this parameter is not given, **'unknown'** is assumed. The meaning of **'unknown'** is processor dependent; our system will create the file if it doesn't exist. If **'scratch'** is given, a temporary file will be created. Temporary files are destroyed at the end of execution. If **'new'** is given, the file must not exist. It will be created for both reading and writing. If **'old'** is given, it is an error for the file not to exist.

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

form= 'formatted' or **'unformatted'.** On UNIX systems, **form='print'** implies **'formatted'** with vertical format control. (See section 3.4 of the text).

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

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

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

6.8.2. CLOSE

close severs the connection between a unit and a file. The unit number must be given. The optional parameters are **iostat=** and **err=** with their usual meanings, and **status=** either **'keep'** or **'delete'.** For scratch files the default is **'delete';** otherwise **'keep'** is the default. **'delete'** means the file will be removed. A simple example is

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

6.8.3. INQUIRE

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

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

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

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

iostat=, err= are as before.

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

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

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

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

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

access= a character variable to which will be assigned the value **'sequential'** if the connection is for sequential I/O, **'direct'** if the connection is for direct I/O, **'unknown'** if not connected.

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

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

form= a character variable to which is assigned the value **'unformatted'** if the file is connected for unformatted I/O, **'formatted'** if the file is connected for formatted I/O, **'print'** for formatted I/O with vertical format control, or **'unknown'** if not connected.

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

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

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

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

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

For information on file permissions, ownership, etc., use the Fortran library routines **stat** and **access**.

For further discussion of the UNIX Fortran I/O system see "Introduction to the f77 I/O Library" [9].

APPENDIX B: References and Bibliography

References

1. *American National Standard Programming Language FORTRAN, ANSI X3.9-1978.* New York: American National Standards Institute, 1978.
2. *USA Standard FORTRAN, USAS X3.9-1966.* New York: United States of America Standards Institute, 1966. Clarified in *Comm. ACM* 12:289 (1969) and *Comm. ACM* 14:628 (1971).
3. Kernighan, B. W., and D. M. Ritchie. *The C Programming Language.* Englewood Cliffs: Prentice-Hall, 1978.
4. Ritchie, D. M. Private communication.
5. Johnson, S. C. "A Portable Compiler: Theory and Practice," *Proceedings of Fifth ACM Symposium on Principles of Programming Languages.* 1978.
6. Feldman, S. I. "An Informal Description of EFL," internal memorandum.
7. Kernighan, B. W. "RATFOR—A Preprocessor for Rational Fortran," *Bell Laboratories Computing Science Technical Report #55.* 1977.
8. Ritchie, D. M. Private communication.
9. Wasley, D. L. "Introduction to the f77 I/O Library", *UNIX Programmer's Manual, Volume 2c.*

Bibliography

The following books or documents describe aspects of Fortran 77. This list cannot pretend to be complete. Certainly no particular endorsement is implied.

1. Brainerd, Walter S., et al. *Fortran 77 Programming.* Harper Row, 1978.
2. Day, A. C. *Compatible Fortran.* Cambridge University Press, 1979.
3. Dock, V. Thomas. *Structured Fortran IV Programming.* West, 1979.
4. Feldman, S. I. "The Programming Language EFL," *Bell Laboratories Technical Report.* June 1979.
5. Hume, J. N., and R. C. Holt. *Programming Fortran 77.* Reston, 1979.
6. Katzan, Harry, Jr. *Fortran 77.* Van Nostrand-Reinhold, 1978.
7. Meissner, Loren P., and Organick, Elliott I. *Fortran 77 Featuring Structured Programming,* Addison-Wesley, 1979.
8. Merchant, Michael J. *ABC's of Fortran Programming.* Wadsworth, 1979.
9. Page, Rex, and Richard Didday. *Fortran 77 for Humans.* West, 1980.
10. Wagener, Jerrold L. *Principles of Fortran 77 Programming.* Wiley, 1980.

Introduction to the f77 I/O Library

David L. Wasley

J. Berkman

University of California, Berkeley
Berkeley, California 94720

PS 1 3

ABSTRACT

The f77 I/O library, libI77.a, includes routines to perform all of the standard types of Fortran input and output specified in the ANSI 1978 Fortran standard. The I/O Library was written originally by Peter J. Weinberger at Bell Labs. Where the original implementation was incomplete, it has been rewritten to more closely implement the standard. Where the standard is vague, we have tried to provide flexibility within the constraints of the UNIX† operating system. A number of logical extensions and enhancements have been provided such as the use of the C stdio library routines to provide efficient buffering for file I/O.

Revised September, 1985

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

Table of Contents

1. Fortran I/O

The requirements of the ANSI standard impose significant overhead on programs that do large amounts of I/O. Formatted I/O can be very "expensive" while direct access binary I/O is usually very efficient. Because of the complexity of Fortran I/O, some general concepts deserve clarification.

1.1. Types of I/O and logical records

There are four forms of I/O: **formatted, unformatted, list directed,** and **namelist.** The last two are related to formatted but do not obey all the rules for formatted I/O. There are two types of "files": **external** and **internal** and two modes of access to files: **direct** and **sequential.** The definition of a logical record depends upon the combination of I/O form, file type, and access mode specified by the Fortran I/O statement.

1.1.1. Direct access external I/O

A logical record in a **direct** access **external** file is a string of bytes of a length specified when the file is opened. Read and write statements must not specify logical records longer than the original record size definition. Shorter logical records are allowed. **Unformatted** direct writes leave the unfilled part of the record undefined. **Formatted** direct writes cause the unfilled record to be padded with blanks.

1.1.2. Sequential access external I/O

Logical records in **sequentially** accessed **external** files may be of arbitrary and variable length. Logical record length for **unformatted** sequential files is determined by the size of items in the iolist. The requirements of this form of I/O cause the external physical record size to be somewhat larger than the logical record size. For **formatted** write statements, logical record length is determined by the format statement interacting with the iolist at execution time. The "newline" character is the logical record delimiter. Formatted sequential access causes one or more logical records ending with "newline" characters to be read or written.

1.1.3. List directed and namelist sequential external I/O

Logical record length for **list directed** and **namelist** I/O is relatively meaningless. On output, the record length is dependent on the magnitude of the data items. On input, the record length is determined by the data types and the file contents. By ANSI definition, a slash, "/", terminates execution of a list directed input operation. Namelist input is terminated by "&end" or "$end" (depending on whether the character before the namelist name was "&" or "$").

1.1.4. Internal I/O

The logical record length for an **internal** read or write is the length of the character variable or array element. Thus a simple character variable is a single logical record. A character variable array is similar to a fixed length direct access file, and obeys the same rules. **Unformatted** and **namelist** I/O are not allowed on "internal" files.

1.2. I/O execution

Note that each execution of a Fortran **unformatted** I/O statement causes a single logical record to be read or written. Each execution of a Fortran **formatted** I/O statement causes one or more logical records to be read or written.

A slash, "/", will terminate assignment of values to the input list during **list directed** input and the remainder of the current input line is skipped. The standard is rather vague on this point but seems to require that a new external logical record be found at the start of any formatted input. Therefore data following the slash is ignored and may be used to comment the data file.

Direct access list directed I/O is not allowed. **Unformatted internal** I/O is not allowed. **Namelist** I/O is allowed only with **external sequential** files. All other flavors of I/O are allowed, although some are not part of the ANSI standard.

Any I/O statement may include an **err=** clause to specify an alternative branch to be taken on errors and/or an **iostat=** clause to return the specific error code. Any error detected during I/O processing will cause the program to abort unless either **err=** or **iostat=** has been specificed in the program. Read statements may include **end=** to branch on end-of-file. The end-of-file indication for that logical unit may be reset with a **backspace** statement. File position and the value of I/O list items is undefined following an error.

2. Implementation details

Some details of the current implementation may be useful in understanding constraints on Fortran I/O.

PS 3 1

2.1. Number of logical units

Unit numbers must be in the range 0 - 99. The maximum number of logical units that a program may have open at one time is the same as the UNIX system limit, currently 48.

2.2. Standard logical units

By default, logical units 0, 5, and 6 are opened to "stderr", "stdin", and "stdout" respectively. However they can be re-defined with an **open** statement. To preserve error reporting, it is an error to close logical unit 0 although it may be reopened to another file.

If you want to open the default file name for any preconnected logical unit, remember to **close** the unit first. Redefining the standard units may impair normal console I/O. An alternative is to use shell re-direction to externally re-define the above units. To re-define default blank control or format of the standard input or output files, use the **open** statement specifying the unit number and no file name (see § 2.4).

The standard units, 0, 5, and 6, are named internally "stderr", "stdin", and "stdout" respectively. These are not actual file names and can not be used for opening these units. **Inquire** will not return these names and will indicate that the above units are not named unless they have been opened to real files. The names are meant to make error reporting more meaningful.

2.3. Vertical format control

Simple vertical format control is implemented. The logical unit must be opened for sequential access with **form = 'print'** (see § 3.2). Control codes "0" and "1" are replaced in the output file with "\n" and "\f" respectively. The control character "+" is not implemented and, like any other character in the first position of a record written to a "print" file, is dropped. The **form = 'print'** mode does not recognize vertical format control for **direct formatted, list directed,** or **namelist** output.

An alternative is to use the filter *fpr*(1) for vertical format control. It replaces "0" and "1" by "\n" and "\f" respectively, and implements the "+" control code. Unlike **form = 'print'** which drops unrecognized form control characters, *fpr* copies those characters to the output file.

2.4. File names and the open statement

A file name may be specified in an **open** statement for the logical unit. If a logical unit is opened by an **open** statement which does not specify a file name, or it is opened implicitly by the execution of a **read, write,** or **endfile** statement, then the default file name is *fort.N* where *N* is the logical unit number. Before opening the file, the library checks for an environment variable with a name identical to the tail of the file name with periods removed.† If it finds such an environment variable, it uses its value as the actual name of the file. For example, a program containing:

†Periods are deleted because they can not be part of environment variable names in the Bourne shell.

```
open(32,file='/usr/guest/census/data.d')
read(32,100) vec
write(44) vec
```

normally will read from */usr/guest/census/data.d* and write to *fort.44* in the current directory. If the environment variables *datad* and *fort44* are set, e.g.:

```
% setenv datad mydata
% setenv fort44 myout
```

in the C shell or:

```
$ datad=mydata
$ fort44=myout
$ export datad fort44
```

in the Bourne shell, then the program will read from *mydata* and write to *myout.*

An **open** statement need not specify a file name. If it refers to a logical unit that is already open, the **blank**= and **form**= specifiers may be redefined without affecting the current file position. Otherwise, if **status** = **'scratch'** is specified, a temporary file with a name of the form *tmp.FXXXXXX* will be opened, and, by default, will be deleted when closed or during termination of program execution.

It is an error to try to open an existing file with **status** = **'new'** . It is an error to try to open a nonexistent file with **status** = **'old'** . By default, **status** = **'unknown'** will be assumed, and a file will be created if necessary.

By default, files are positioned at their beginning upon opening, but see *fseek*(3f) and *ioinit*(3f) for alternatives. Existing files are never truncated on opening. Sequentially accessed external files are truncated to the current file position on **close**, **backspace**, or **rewind** only if the last access to the file was a write. An **endfile** always causes such files to be truncated to the current file position.

2.5. Format interpretation

Formats which are in format statements are parsed by the compiler; formats in **read**, **write**, and **print** statements are parsed during execution by the I/O library. Upper as well as lower case characters are recognized in format statements and all the alphabetic arguments to the I/O library routines.

If the external representation of a datum is too large for the field width specified, the specified field is filled with asterisks (*). On Ew.dEe output, the exponent field will be filled with asterisks if the exponent representation is too large. This will only happen if "e" is zero (see appendix B).

On output, a real value that is truly zero will display as "0." to distinguish it from a very small non-zero value. If this causes problems for other input systems, the **BZ** edit descriptor may be used to cause the field following the decimal point to be filled with zero's.

Non-destructive tabbing is implemented for both internal and external formatted I/O. Tabbing left or right on output does not affect previously written portions of a record. Tabbing right on output causes unwritten portions of a record to be filled with blanks. Tabbing right off the end of an input logical record is an error. Tabbing left beyond the beginning of an input logical record leaves the input pointer at the beginning of the record. The format specifier **T** must be followed by a positive non-zero number. If it is not, it will have a different meaning (see § 3.1).

Tabbing left requires seek ability on the logical unit. Therefore it is not allowed in I/O to a terminal or pipe. Likewise, nondestructive tabbing in either direction is possible only on a unit that can seek. Otherwise tabbing right or spacing with **X** will write blanks on the output.

2.6. List directed output

In formatting list directed output, the I/O system tries to prevent output lines longer than 80 characters. Each external datum will be separated by two spaces. List directed output of **complex** values includes an appropriate comma. List directed output distinguishes between **real** and **double precision** values and formats them differently. Output of a character string that includes "\n" is

interpreted reasonably by the output system.

2.7. I/O errors

If I/O errors are not trapped by the user's program an appropriate error message will be written to "stderr" before aborting. An error number will be printed in "[ ]" along with a brief error message showing the logical unit and I/O state. Error numbers < 100 refer to UNIX errors, and are described in the introduction to chapter 2 of the UNIX Programmer's Manual. Error numbers ≥ 100 come from the I/O library, and are described further in the appendix to this writeup‡. For internal I/O, part of the string will be printed with "|" at the current position in the string. For external I/O, part of the current record will be displayed if the error was caused during reading from a file that can backspace.

3. Non-"ANSI Standard" extensions

Several extensions have been added to the I/O system to provide for functions omitted or poorly defined in the standard. Programmers should be aware that these are non-portable.

3.1. Format specifiers

B is an acceptable edit control specifier. It causes return to the logical unit's default mode of blank interpretation. This is consistent with **S** which returns to default sign control.

P by itself is equivalent to **0P** . It resets the scale factor to the default value, 0.

The form of the Ew.dEe format specifier has been extended to **D** also. The form Ew.d.e is allowed but is not standard. The "e" field specifies the minimum number of digits or spaces in the exponent field on output. If the value of the exponent is too large, the exponent notation **e** or **d** will be dropped from the output to allow one more character position. If this is still not adequate, the "e" field will be filled with asterisks (*). The default value for "e" is 2.

An additional form of tab control specification has been added. The ANSI standard forms **TRn**, **TLn**, and **Tn** are supported where *n* is a positive non-zero number. If **T** or **nT** is specified, tabbing will be to the next (or n-th) 8-column tab stop. Thus columns of alphanumerics can be lined up without counting.

A format control specifier has been added to suppress the newline at the end of the last record of a formatted sequential write. The specifier is a dollar sign ($). It is constrained by the same rules as the colon (:). It is used typically for console prompts. For example:

```
write (*, "('enter value for x: ',$)")
read (*,*) x
```

Radices other than 10 can be specified for formatted integer I/O conversion. The specifier is patterned after **P**, the scale factor for floating point conversion. It remains in effect until another radix is specified or format interpretation is complete. The specifier is defined as **[n]R** where $2 \leq n \leq 36$. If *n* is omitted, the default decimal radix is restored.

The format specifier **Om.n** may be used for an octal conversion; it is equivalent to **8R,Im.n,10R**. Similarly, **Zm.n** is equivalent to **16R,Im.n,10R** and may be used for an hexadecimal conversion;

In conjunction with the above, a sign control specifier has been added to cause integer values to be interpreted as unsigned during output conversion. The specifier is **SU** and remains in effect until another sign control specifier is encountered, or format interpretation is complete.† Radix and

‡ On many systems, these are also available in *help f77 io_err_msgs*.

†Note: Unsigned integer values greater than (2**31 - 1), can be read and written using SU. However they can not be used in computations because Fortran uses signed arithmetic and such values appear to the arithmetic unit as negative numbers.

"unsigned" specifiers could be used to format a hexadecimal dump, as follows:

```
2000  format ( SU, 8Z10.8 )
```

3.2. Print files

The ANSI standard is ambiguous regarding the definition of a "print" file. Since UNIX has no default "print" file, an additional **form=** specifier is now recognized in the **open** statement. Specifying **form = 'print'** implies **formatted** and enables vertical format control for that logical unit (see §2.3). Vertical format control is interpreted only on sequential formatted writes to a "print" file.

The **inquire** statement will return **print** in the **form=** string variable for logical units opened as "print" files. It will return -1 for the unit number of an unconnected file.

If a logical unit is already open, an **open** statement including the **form=** option or the **blank=** option will do nothing but re-define those options. This instance of the **open** statement need not include the file name, and must not include a file name if **unit=** refers to a standard input or output. Therefore, to re-define the standard output as a "print" file, use:

```
open (unit=6, form='print')
```

3.3. Scratch files

A **close** statement with **status = 'keep'** may be specified for temporary files. This is the default for all other files. Remember to get the scratch file's real name, using **inquire** , if you want to re-open it later.

3.4. List directed I/O

List directed read has been modified to allow tab characters wherever blanks are allowed. It also allows input of a string not enclosed in quotes. The string must not start with a digit or quote, and can not contain any separators (",", "/", blank or tab). A newline will terminate the string unless escaped with \. Any string not meeting the above restrictions must be enclosed in quotes (" " " or " ' ").

Internal list directed I/O has been implemented. During internal list reads, bytes are consumed until the iolist is satisfied, or the "end-of-file" is reached. During internal list writes, records are filled until the iolist is satisfied. The length of an internal array element should be at least 20 bytes to avoid logical record overflow when writing double precision values. Internal list read was implemented to make command line decoding easier. Internal list write should be avoided.

3.5. Namelist I/O

Namelist I/O is a common extension in Fortran systems. The **f77** version was designed to be compatible with other vendors versions; it is described in "A Portable Fortran 77 Compiler", by Feldman and Weinberger, August, 1985.

4. Running older programs

Traditional Fortran environments usually assume carriage control on all logical units, usually interpret blank spaces on input as "0"s, and often provide attachment of global file names to logical units at run time. There are several routines in the I/O library to provide these functions.

4.1. Traditional unit control parameters

If a program reads and writes only units 5 and 6, then including **-I66** in the f77 command will cause carriage control to be interpreted on output and cause blanks to be zeros on input without further modification of the program. If this is not adequate, the routine *ioinit*(3f) can be called to specify control parameters separately, including whether files should be positioned at their beginning or end upon opening.

4.2. Ioinit()

Ioinit(3f) can be used to attach logical units to specific files at run time, and to set global parameters for the I/O system. It will look for names of a user specified form in the environment and open the corresponding logical unit for **sequential formatted** I/O. Names must be of the form **PREFIX***nn* where **PREFIX** is specified in the call to *ioinit* and *nn* is the logical unit to be opened. Unit numbers < 10 must include the leading "0".

Ioinit should prove adequate for most programs as written. However, it is written in Fortran-77 specifically so that it may serve as an example for similar user-supplied routines. A copy may be retrieved by "ar x /usr/lib/libU77.a ioinit.f". See §2.4 for another way to override program file names through environment variables.

5. Magnetic tape I/O

Because the I/O library uses stdio buffering, reading or writing magnetic tapes should be done with great caution, or avoided if possible. A set of routines has been provided to read and write arbitrary sized buffers to or from tape directly. The buffer must be a **character** object. **Internal** I/O can be used to fill or interpret the buffer. These routines do not use normal Fortran I/O processing and do not obey Fortran I/O rules. See *topen*(3f).

6. Caveat Programmer

The I/O library is extremely complex yet we believe there are few bugs left. We've tried to make the system as correct as possible according to the ANSI X3.9-1978 document and keep it compatible with the UNIX file system. Exceptions to the standard are noted in appendix B.

Appendix A

I/O Library Error Messages

The following error messages are generated by the I/O library. The error numbers are returned in the **iostat=** variable. Error numbers < 100 are generated by the UNIX kernel. See the introduction to chapter 2 of the UNIX Programmers Manual for their description.

100 *error in format*
See error message output for the location of the error in the format. Can be caused by more than 10 levels of nested parentheses, or an extremely long format statement.

101 *illegal unit number*
It is illegal to close logical unit 0. Unit numbers must be between 0 and 99 inclusive.

102 *formatted i/o not allowed*
The logical unit was opened for unformatted I/O.

103 *unformatted i/o not allowed*
The logical unit was opened for formatted I/O.

104 *direct i/o not allowed*
The logical unit was opened for sequential access, or the logical record length was specified as 0.

105 *sequential i/o not allowed*
The logical unit was opened for direct access I/O.

106 *can't backspace file*
The file associated with the logical unit can't seek. May be a device or a pipe.

107 *off beginning of record*
The format specified a left tab beyond the beginning of an internal input record.

108 *can't stat file*
The system can't return status information about the file. Perhaps the directory is unreadable.

109 *no * after repeat count*
Repeat counts in list directed I/O must be followed by an * with no blank spaces.

110 *off end of record*
A formatted write tried to go beyond the logical end-of-record. An unformatted read or write will also cause this.

111 *truncation failed*
The truncation of an external sequential file on **close**, **backspace**, **rewind**, or **endfile** failed.

112 *incomprehensible list input*
List input has to be just right.

113 *out of free space*
The library dynamically creates buffers for internal use. You ran out of memory for this. Your program is too big!

114 *unit not connected*
The logical unit was not open.

115 *invalid data for integer format term*
Only spaces, a leading sign and digits are allowed.

116 *invalid data for logical format term*
Legal input consists of spaces (optional), a period (optional), and then a "t", "T", "f", or "F".

117 *'new' file exists*
You tried to open an existing file with "status='new'".

118 *can't find 'old' file*
You tried to open a non-existent file with "status='old'".

119 *opening too many files or unknown system error*
Either you are trying to open too many files simultaneously or there has been an undetected system error.

120 *requires seek ability*
Direct access requires seek ability. Sequential unformatted I/O requires seek ability on the file due to the special data structure required. Tabbing left also requires seek ability.

121 *illegal argument*
Certain arguments to **open**, etc. will be checked for legitimacy. Often only non-default forms are looked for.

122 *negative repeat count*
The repeat count for list directed input must be a positive integer.

123 *illegal operation for unit*
An operation was requested for a device associated with the logical unit which was not possible. This error is returned by the tape I/O routines if attempting to read past end-of-tape, etc.

124 *invalid data for d, e, f or g format term*
Input data must be legal.

125 *illegal input for namelist*
Column one of input is ignored, the namelist name must match, the variables must be in the namelist, and the data must be of the right type.

Appendix B

Exceptions to the ANSI Standard

A few exceptions to the ANSI standard remain.

Vertical format control

The "+" carriage control specifier is not fully implemented (see §2.3). It would be difficult to implement it correctly and still provide UNIX-like file I/O.

Furthermore, the carriage control implementation is asymmetrical. A file written with carriage control interpretation can not be read again with the same characters in column 1.

An alternative to interpreting carriage control internally is to run the output file through a "Fortran output filter" before printing. This filter could recognize a much broader range of carriage control and include terminal dependent processing. One such filter is *fpr*(1).

Default files

Files created by default use of **endfile** statements are opened for **sequential formatted** access. There is no way to redefine such a file to allow **direct** or **unformatted** access.

Lower case strings

It is not clear if the ANSI standard requires internally generated strings to be upper case or not. As currently written, the **inquire** statement will return lower case strings for any alphanumeric data.

Exponent representation on Ew.dEe output

If the field width for the exponent is too small, the standard allows dropping the exponent character but only if the exponent is > 99. This system does not enforce that restriction. Further, the standard implies that the entire field, "w", should be filled with asterisks if the exponent can not be displayed. This system fills only the exponent field in the above case since that is more diagnostic.

Pre-connection of files

The standard says units must be pre-connected to files before the program starts or must be explicitly opened. Instead, the I/O library connects the unit to a file on its first use in a **read, write, print,** or **endfile** statement. Thus **inquire** by unit can not tell prior to a unit number use the characteristics or name of the file corresponding to a unit.

Berkeley Pascal User's Manual
Version 3.1 - April 1986

William N. Joy‡, Susan L. Graham, Charles B. Haley‡,
Marshall Kirk McKusick, and Peter B. Kessler‡

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

ABSTRACT

Berkeley Pascal is designed for interactive instructional use and runs on the PDP/11 and VAX/11 computers. Interpretive code is produced, providing fast translation at the expense of slower execution speed. There is also a fully compatible compiler for the VAX/11. An execution profiler and Wirth's cross reference program are also available with the system.

The system supports full Pascal. The language accepted is 'standard' Pascal, and a small number of extensions. There is an option to suppress the extensions. The extensions include a separate compilation facility and the ability to link to object modules produced from other source languages.

The *User's Manual* gives a list of sources relating to the UNIX† system, the Pascal language, and the Berkeley Pascal system. Basic usage examples are provided for the Pascal components *pi*, *px*, *pix*, *pc*, and *pxp*. Errors commonly encountered in these programs are discussed. Details are given of special considerations due to the interactive implementation. A number of examples are provided including many dealing with input/output. An appendix supplements Wirth's *Pascal Report* to form the full definition of the Berkeley implementation of the language.

Introduction

The Berkeley Pascal *User's Manual* consists of five major sections and an appendix. In section 1 we give sources of information about UNIX, about the programming language Pascal, and about the Berkeley implementation of the language. Section 2 introduces the Berkeley implementation and provides a number of tutorial examples. Section 3 discusses the error diagnostics produced by the translators *pc* and *pi*, and the runtime interpreter *px*. Section 4 describes input/output with special attention given to features of the interactive implementation and to features unique to UNIX. Section 5 gives details on the components of the system and explanation of all relevant options. The *User's Manual* concludes with an appendix to Wirth's *Pascal Report* with which it forms a precise definition of the implementation.

---

‡Author's current addresses: William Joy: Sun Microsystems, 2550 Garcia Ave., Mountain View, CA 94043; Charles Haley: S & B Associates, 1110 Centennial Ave., Piscataway, NJ 08854; Peter Kessler: Xerox Research Park, Palo Alto, CA
† UNIX is a trademark of AT&T Bell Laboratories.

History of the implementation

The first Berkeley system was written by Ken Thompson in early 1976. The main features of the present system were implemented by Charles Haley and William Joy during the latter half of 1976. Earlier versions of this system have been in use since January, 1977.

The system was moved to the VAX-11 by Peter Kessler and Kirk McKusick with the porting of the interpreter in the spring of 1979, and the implementation of the compiler in the summer of 1980.

1. Sources of information

This section lists the resources available for information about general features of UNIX, text editing, the Pascal language, and the Berkeley Pascal implementation, concluding with a list of references. The available documents include both so-called standard documents – those distributed with all UNIX system – and documents (such as this one) written at Berkeley.

1.1. Where to get documentation

Current documentation for most of the UNIX system is available "on line" at your terminal. Details on getting such documentation interactively are given in section 1.3.

1.2. Documentation describing UNIX

The following documents are those recommended as tutorial and reference material about the UNIX system. We give the documents with the introductory and tutorial materials first, the reference materials last.

UNIX For Beginners – Second Edition

This document is the basic tutorial for UNIX available with the standard system.

Communicating with UNIX

This is also a basic tutorial on the system and assumes no previous familiarity with computers; it was written at Berkeley.

An introduction to the C shell

This document introduces *csh,* the shell in common use at Berkeley, and provides a good deal of general description about the way in which the system functions. It provides a useful glossary of terms used in discussing the system.

UNIX Programmer's Manual

This manual is the major source of details on the components of the UNIX system. It consists of an Introduction, a permuted index, and eight command sections. Section 1 consists of descriptions of most of the "commands" of UNIX. Most of the other sections have limited relevance to the user of Berkeley Pascal, being of interest mainly to system programmers.

UNIX documentation often refers the reader to sections of the manual. Such a reference consists of a command name and a section number or name. An example of such a reference would be: *ed* (1). Here *ed* is a command name – the standard UNIX text editor, and '(1)' indicates that its documentation is in section 1 of the manual.

The pieces of the Berkeley Pascal system are *pi* (1), *px* (1), the combined Pascal translator and interpretive executor *pix* (1), the Pascal compiler *pc* (1), the Pascal execution profiler *pxp* (1), and the Pascal cross-reference generator *pxref* (1).

It is possible to obtain a copy of a manual section by using the *man* (1) command. To get the Pascal documentation just described one could issue the command:

% man pi

to the shell. The user input here is shown in **bold face**; the '% ', which was printed by the shell as a prompt, is not. Similarly the command:

% man man

asks the *man* command to describe itself.

1.3. Text editing documents

The following documents introduce the various UNIX text editors. Most Berkeley users use a version of the text editor *ex;* either *edit,* which is a version of *ex* for new and casual users, *ex* itself, or *vi* (visual) which focuses on the display editing portion of *ex.*

A Tutorial Introduction to the UNIX Text Editor

This document, written by Brian Kernighan of Bell Laboratories, is a tutorial for the standard UNIX text editor *ed.* It introduces you to the basics of text editing, and provides enough information to meet day-to-day editing needs, for *ed* users.

Edit: A tutorial

This introduces the use of *edit,* an editor similar to *ed* which provides a more hospitable environment for beginning users.

Ex/edit Command Summary

This summarizes the features of the editors *ex* and *edit* in a concise form. If you have used a line oriented editor before this summary alone may be enough to get you started.

Ex Reference Manual – Version 3.7

A complete reference on the features of *ex* and *edit.*

An Introduction to Display Editing with Vi

Vi is a display oriented text editor. It can be used on most any CRT terminal, and uses the screen as a window into the file you are editing. Changes you make to the file are reflected in what you see. This manual serves both as an introduction to editing with *vi* and a reference manual.

Vi Quick Reference

This reference card is a handy quick guide to *vi;* you should get one when you get the introduction to *vi.*

1.4. Pascal documents – The language

This section describes the documents on the Pascal language which are likely to be most useful to the Berkeley Pascal user. Complete references for these documents are given in section 1.7.

Pascal User Manual

By Kathleen Jensen and Niklaus Wirth, the *User Manual* provides a tutorial introduction to the features of the language Pascal, and serves as an excellent quick-reference to the language. The reader with no familiarity with Algol-like languages may prefer one of the Pascal text books listed below, as they provide more examples and explanation. Particularly important here are pages 116-118 which define the syntax of the language. Sections 13 and 14 and Appendix F pertain only to the 6000-3.4 implementation of Pascal.

Pascal Report

By Niklaus Wirth, this document is bound with the *User Manual.* It is the guiding reference for implementors and the fundamental definition of the language. Some programmers find this report too concise to be of practical use, preferring the *User Manual* as a reference.

Books on Pascal

Several good books which teach Pascal or use it as a medium are available. The books by Wirth *Systematic Programming* and *Algorithms + Data Structures = Programs* use Pascal as a vehicle for teaching programming and data structure concepts respectively. They are both recommended. Other books on Pascal are listed in the references below.

1.5. Pascal documents – The Berkeley Implementation

This section describes the documentation which is available describing the Berkeley implementation of Pascal.

User's Manual

The document you are reading is the *User's Manual* for Berkeley Pascal. We often refer the reader to the Jensen-Wirth *User Manual* mentioned above, a different document with a similar name.

Manual sections

The sections relating to Pascal in the *UNIX Programmer's Manual* are *pix* (1), *pi* (1), *pc* (1), *px* (1), *pxp* (1), and *pxref* (1). These sections give a description of each program, summarize the available options, indicate files used by the program, give basic information on the diagnostics produced and include a list of known bugs.

Implementation notes

For those interested in the internal organization of the Berkeley Pascal system there are a series of *Implementation Notes* describing these details. The *Berkeley Pascal PXP Implementation Notes* describe the Pascal interpreter *px*; and the *Berkeley Pascal PX Implementation Notes* describe the structure of the execution profiler *pxp*.

1.6. References

UNIX Documents

Communicating With UNIX
Computer Center
University of California, Berkeley
January, 1978.

Ricki Blau and James Joyce
Edit: a tutorial
UNIX User's Supplementary Documents (USD), 14
University of California, Berkeley, CA. 94720
April, 1986.

Ex/edit Command Summary
Computer Center
University of California, Berkeley
August, 1978.

William Joy
Ex Reference Manual - Version 3.7
UNIX User's Supplementary Documents (USD), 16
University of California, Berkeley, CA. 94720
April, 1986.

William Joy
An Introduction to Display Editing with Vi
UNIX User's Supplementary Documents (USD), 15
University of California, Berkeley, CA. 94720
April, 1986.

William Joy
An Introduction to the C shell (Revised)
UNIX User's Supplementary Documents (USD), 4
University of California, Berkeley, CA. 94720
April, 1986.

Brian W. Kernighan
UNIX for Beginners - Second Edition
UNIX User's Supplementary Documents (USD), 1
University of California, Berkeley, CA. 94720
April, 1986.

Brian W. Kernighan
A Tutorial Introduction to the UNIX Text Editor
UNIX User's Supplementary Documents (USD), 12
University of California, Berkeley, CA. 94720
April, 1986.

Dennis M. Ritchie and Ken Thompson
The UNIX Time Sharing System
Reprinted from Communications of the ACM July 1974 in
UNIX Programmer's Supplementary Documents, Volume 2 (PS2), 1
University of California, Berkeley, CA. 94720
April, 1986.

Pascal Language Documents

Cooper and Clancy
Oh! Pascal!, 2nd Edition
W. W. Norton & Company, Inc.
500 Fifth Ave., NY, NY. 10110
1985, 475 pp.

Cooper
Standard Pascal User Reference Manual
W. W. Norton & Company, Inc.
500 Fifth Ave., NY, NY. 10110
1983, 176 pp.

Kathleen Jensen and Niklaus Wirth
Pascal - User Manual and Report
Springer-Verlag, New York.
1975, 167 pp.

Niklaus Wirth
Algorithms + Data structures = Programs
Prentice-Hall, New York.
1976, 366 pp.

Berkeley Pascal documents

The following documents are available from the Computer Center Library at the University of California, Berkeley.

William N. Joy
Berkeley Pascal PX Implementation Notes
Version 1.1, April 1979.
(Vax-11 Version 2.0 By Kirk McKusick, December, 1979)

William N. Joy
Berkeley Pascal PXP Implementation Notes
Version 1.1, April 1979.

2. Basic UNIX Pascal

The following sections explain the basics of using Berkeley Pascal. In examples here we use the text editor *ex* (1). Users of the text editor *ed* should have little trouble following these examples, as *ex* is similar to *ed*. We use *ex* because it allows us to make clearer examples.† The new UNIX user will find it helpful to read one of the text editor documents described in section 1.4 before continuing with this section.

2.1. A first program

To prepare a program for Berkeley Pascal we first need to have an account on UNIX and to 'login' to the system on this account. These procedures are described in the documents *Communicating with UNIX* and *UNIX for Beginners*.

Once we are logged in we need to choose a name for our program; let us call it 'first' as this is the first example. We must also choose a name for the file in which the program will be stored. The Berkeley Pascal system requires that programs reside in files which have names ending with the sequence '.p' so we will call our file 'first.p'.

A sample editing session to create this file would begin:

```
% ex first.p
"first.p" [New file]
:
```

We didn't expect the file to exist, so the error diagnostic doesn't bother us. The editor now knows the name of the file we are creating. The ':' prompt indicates that it is ready for command input. We can add the text for our program using the 'append' command as follows.

```
:append
```

† Users with CRT terminals should find the editor *vi* more pleasant to use; we do not show its use here because its display oriented nature makes it difficult to illustrate.

```
program first(output)
begin
    writeln('Hello, world!')
end.
.
:
```

The line containing the single '.' character here indicated the end of the appended text. The ':' prompt indicates that *ex* is ready for another command. As the editor operates in a temporary work space we must now store the contents of this work space in the file 'first.p' so we can use the Pascal translator and executor *pix* on it.

```
:write
"first.p" [New file] 4 lines, 59 characters
:quit
%
```

We wrote out the file from the edit buffer here with the 'write' command, and *ex* indicated the number of lines and characters written. We then quit the editor, and now have a prompt from the shell.‡

We are ready to try to translate and execute our program.

```
% pix first.p
Wed May  7 14:56 1986  first.p:
    2  begin
e -----↑--- Inserted ';'
Execution begins...
Hello, world!
Execution terminated.

1 statements executed in 0.00 seconds cpu time.
%
```

The translator first printed a syntax error diagnostic. The number 2 here indicates that the rest of the line is an image of the second line of our program. The translator is saying that it expected to find a ';' before the keyword **begin** on this line. If we look at the Pascal syntax charts in the Jensen-Wirth *User Manual*, or at some of the sample programs therein, we will see that we have omitted the terminating ';' of the **program** statement on the first line of our program.

One other thing to notice about the error diagnostic is the letter 'e' at the beginning. It stands for 'error', indicating that our input was not legal Pascal. The fact that it is an 'e' rather than an 'E' indicates that the translator managed to recover from this error well enough that generation of code and execution could take place. Execution is possible whenever no fatal 'E' errors occur during translation. The other classes of diagnostics are 'w' warnings, which do not necessarily indicate errors in the program, but point out inconsistencies which are likely to be due to program bugs, and 's' standard-Pascal violations.†

After completing the translation of the program to interpretive code, the Pascal system indicates that execution of the translated program began. The output from the execution of the program then appeared. At program termination, the Pascal runtime system indicated the number of statements executed, and the amount of cpu time used, with the resolution of the

‡ Our examples here assume you are using *csh*.

†The standard Pascal warnings occur only when the associated s translator option is enabled. The s option is discussed in sections 5.1 and A.6 below. Warning diagnostics are discussed at the end of section 3.2, the associated w option is described in section 5.2.

latter being 1/60'th of a second.

Let us now fix the error in the program and translate it to a permanent object code file *obj* using *pi*. The program *pi* translates Pascal programs but stores the object code instead of executing it‡.

```
% ex first.p
"first.p" 4 lines, 59 characters
:1 print
program first(output)
:s/$/;
program first(output);
:write
"first.p" 4 lines, 60 characters
:quit
% pi first.p
%
```

If we now use the UNIX *ls* list files command we can see what files we have:

```
% ls
first.p
obj
%
```


The file 'obj' here contains the Pascal interpreter code. We can execute this by typing:

```
% px obj
Hello, world!

1 statements executed in 0.00 seconds cpu time.
%
```

Alternatively, the command:

```
% obj
```

will have the same effect. Some examples of different ways to execute the program follow.

```
% px
Hello, world!

1 statements executed in 0.00 seconds cpu time.
% pi -p first.p
% px obj
Hello, world!
% pix -p first.p
Hello, world!
%
```

Note that *px* will assume that 'obj' is the file we wish to execute if we don't tell it otherwise. The last two translations use the **-p** no-post-mortem option to eliminate execution

‡This script indicates some other useful approaches to debugging Pascal programs. As in *ed* we can shorten commands in *ex* to an initial prefix of the command name as we did with the *substitute* command here. We have also used the '!' shell escape command here to execute other commands with a shell without leaving the editor.

statistics and 'Execution begins' and 'Execution terminated' messages. See section 5.2 for more details. If we now look at the files in our directory we will see:

```
% ls
first.p
obj
%
```

We can give our object program a name other than 'obj' by using the move command *mv* (1). Thus to name our program 'hello':

```
% mv obj hello
% hello
Hello, world!
% ls
first.p
hello
%
```

Finally we can get rid of the Pascal object code by using the *rm* (1) remove file command, e.g.:

```
% rm hello
% ls
first.p
%
```


For small programs which are being developed *pix* tends to be more convenient to use than *pi* and *px*. Except for absence of the *obj* file after a *pix* run, a *pix* command is equivalent to a *pi* command followed by a *px* command. For larger programs, where a number of runs testing different parts of the program are to be made, *pi* is useful as this *obj* file can be executed any desired number of times.

2.2. A larger program

Suppose that we have used the editor to put a larger program in the file 'bigger.p'. We can list this program with line numbers by using the program *cat*-n i.e.:

```
% cat -n bigger.p
     1  (*
     2   * Graphic representation of a function
     3   *    f(x) = exp(-x) * sin(2 * pi * x)
     4   *)
     5  program graph1(output);
     6  const
     7            d = 0.0625;  (* 1/16, 16 lines for interval [x, x+1] *)
     8            s = 32;      (* 32 character width for interval [x, x+1]
     9            h = 34;      (* Character position of x-axis *)
    10            c = 6.28138; (* 2 * pi *)
    11            lim = 32;
    12  var
    13            x, y: real;
    14            i, n: integer;
    15  begin
    16            for i := 0 to lim begin
    17                      x := d / i;
```

```
    18                      y := exp(-x9 * sin(i * x);
    19                      n := Round(s * y) + h;
    20                      repeat
    21                              write(' ');
    22                              n := n - 1
    23                      writeln('*')
    24  end.
%
```

This program is similar to program 4.9 on page 30 of the Jensen-Wirth *User Manual*. A number of problems have been introduced into this example for pedagogical reasons.

If we attempt to translate and execute the program using *pix* we get the following response:

```
% pix bigger.p
Wed May  7 14:56 1986  bigger.p:
    9       h = 34;      (* Character position of x-axis *)
w ------------------------------↑ ------ (* in a (* ... *) comment
   16       for i := 0 to lim begin
e ------------------------------------↑ ------ Inserted keyword do
   18               y := exp(-x9 * sin(i * x);
E ------------------------------------↑ ------ Undefined variable
e ------------------------------------------------↑ ------ Inserted ')'
   19               n := Round(s * y) + h;
E ----------------------------↑ ------ Undefined function
E ------------------------------------------------↑ ------ Undefined variable
   23               writeln('*')
e ----------------------↑ ------ Inserted ';'
   24  end.
E ------↑ ------ Expected keyword until
E ----------↑ ------ Malformed declaration
E ----------↑ ------ Unexpected end-of-file - QUIT
Execution suppressed due to compilation errors
%
```

Since there were fatal 'E' errors in our program, no code was generated and execution was necessarily suppressed. One thing which would be useful at this point is a listing of the program with the error messages. We can get this by using the command:

```
% pi -l bigger.p
```

There is no point in using *pix* here, since we know there are fatal errors in the program. This command will produce the output at our terminal. If we are at a terminal which does not produce a hard copy we may wish to print this listing off-line on a line printer. We can do this with the command:

```
% pi -l bigger.p | lpr
```

In the next few sections we will illustrate various aspects of the Berkeley Pascal system by correcting this program.

2.3. Correcting the first errors

Most of the errors which occurred in this program were *syntactic* errors, those in the format and structure of the program rather than its content. Syntax errors are flagged by printing the offending line, and then a line which flags the location at which an error was

detected. The flag line also gives an explanation stating either a possible cause of the error, a simple action which can be taken to recover from the error so as to be able to continue the analysis, a symbol which was expected at the point of error, or an indication that the input was 'malformed'. In the last case, the recovery may skip ahead in the input to a point where analysis of the program can continue.

In this example, the first error diagnostic indicates that the translator detected a comment within a comment. While this is not considered an error in 'standard' Pascal, it usually corresponds to an error in the program which is being translated. In this case, we have accidentally omitted the trailing '*)' of the comment on line 8. We can begin an editor session to correct this problem by doing:

```
% ex bigger.p
"bigger.p" 24 lines, 512 characters
:8s/$/ *)
        s = 32;      (* 32 character width for interval [x, x+1] *)
:
```

The second diagnostic, given after line 16, indicates that the keyword **do** was expected before the keyword **begin** in the **for** statement. If we examine the *statement* syntax chart on page 118 of the Jensen-Wirth *User Manual* we will discover that **do** is a necessary part of the **for** statement. Similarly, we could have referred to section C.3 of the Jensen-Wirth *User Manual* to learn about the **for** statement and gotten the same information there. It is often useful to refer to these syntax charts and to the relevant sections of this book.

We can correct this problem by first scanning for the keyword **for** in the file and then substituting the keyword **do** to appear in front of the keyword **begin** there. Thus:

```
:/for
        for i := 0 to lim begin
:s/begin/do &
        for i := 0 to lim do begin
:
```

The next error in the program is easy to pinpoint. On line 18, we didn't hit the shift key and got a '9' instead of a ')'. The translator diagnosed that 'x9' was an undefined variable and, later, that a ')' was missing in the statement. It should be stressed that *pi* is not suggesting that you should insert a ')' before the ';'. It is only indicating that making this change will help it to be able to continue analyzing the program so as to be able to diagnose further errors. You must then determine the true cause of the error and make the appropriate correction to the source text.

This error also illustrates the fact that one error in the input may lead to multiple error diagnostics. *Pi* attempts to give only one diagnostic for each error, but single errors in the input sometimes appear to be more than one error. It is also the case that *pi* may not detect an error when it occurs, but may detect it later in the input. This would have happened in this example if we had typed 'x' instead of 'x9'.

The translator next detected, on line 19, that the function *Round* and the variable *h* were undefined. It does not know about *Round* because Berkeley Pascal normally distinguishes between upper and lower case.† On UNIX lower-case is preferred‡, and all keywords and built-in **procedure** and **function** names are composed of lower-case letters, just as they are in the Jensen-Wirth *Pascal Report*. Thus we need to use the function *round* here. As far as *h* is concerned, we can see why it is undefined if we look back to line 9 and note that its

†In "standard" Pascal no distinction is made based on case.
‡One good reason for using lower-case is that it is easier to type.

definition was lost in the non-terminated comment. This diagnostic need not, therefore, concern us.

The next error which occurred in the program caused the translator to insert a ';' before the statement calling *writeln* on line 23. If we examine the program around the point of error we will see that the actual error is that the keyword **until** and an associated expression have been omitted here. Note that the diagnostic from the translator does not indicate the actual error, and is somewhat misleading. The translator made the correction which seemed to be most plausible. As the omission of a ';' character is a common mistake, the translator chose to indicate this as a possible fix here. It later detected that the keyword **until** was missing, but not until it saw the keyword **end** on line 24. The combination of these diagnostics indicate to us the true problem.

The final syntactic error message indicates that the translator needed an **end** keyword to match the **begin** at line 15. Since the **end** at line 24 is supposed to match this **begin**, we can infer that another **begin** must have been mismatched, and have matched this **end**. Thus we see that we need an **end** to match the **begin** at line 16, and to appear before the final **end**. We can make these corrections:

```
:/x9/s//x)
            y := exp(-x) * sin(i * x);
:+s/Round/round
            n := round(s * y) + h;
:/write
                write(' ');
:/
            writeln('*')
:insert
.           until n = 0;
.
:$
end.
:insert
    end
.
:
```

At the end of each **procedure** or **function** and the end of the **program** the translator summarizes references to undefined variables and improper usages of variables. It also gives warnings about potential errors. In our program, the summary errors do not indicate any further problems but the warning that *c* is unused is somewhat suspicious. Examining the program we see that the constant was intended to be used in the expression which is an argument to *sin*, so we can correct this expression, and translate the program. We have now made a correction for each diagnosed error in our program.

```
:?i ?s//c /
            y := exp(-x) * sin(c * x);
:write
"bigger.p" 26 lines, 538 characters
:quit
% pi bigger.p
%
```

It should be noted that the translator suppresses warning diagnostics for a particular **procedure**, **function** or the main **program** when it finds severe syntax errors in that part of the source text. This is to prevent possibly confusing and incorrect warning diagnostics from being produced. Thus these warning diagnostics may not appear in a program with bad

syntax errors until these errors are corrected.

We are now ready to execute our program for the first time. We will do so in the next section after giving a listing of the corrected program for reference purposes.

```
% cat -n bigger.p
     1  (*
     2   * Graphic representation of a function
     3   *    f(x) = exp(-x) * sin(2 * pi * x)
     4   *)
     5  program graph1(output);
     6  const
     7          d = 0.0625;  (* 1/16, 16 lines for interval [x, x+1] *)
     8          s = 32;      (* 32 character width for interval [x, x+1] *)
     9          h = 34;      (* Character position of x-axis *)
    10          c = 6.28138; (* 2 * pi *)
    11          lim = 32;
    12  var
    13          x, y: real;
    14          i, n: integer;
    15  begin
    16          for i := 0 to lim do begin
    17                  x := d / i;
    18                  y := exp(-x) * sin(c * x);
    19                  n := round(s * y) + h;
    20                  repeat
    21                          write(' ');
    22                          n := n - 1
    23                  until n = 0;
    24                  writeln('*')
    25          end
    26  end.
%
```

2.4. Executing the second example

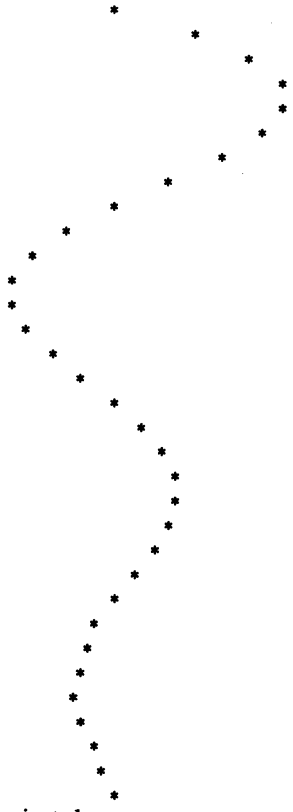
We are now ready to execute the second example. The following output was produced by our first run.

```
% px
Execution begins...

Real division by zero

        Error in "graph1"+2 near line 17.
Execution terminated abnormally.

2 statements executed in 0.00 seconds cpu time.
%
```

Here the interpreter is presenting us with a runtime error diagnostic. It detected a 'division by zero' at line 17. Examining line 17, we see that we have written the statement 'x := d / i' instead of 'x := d * i'. We can correct this and rerun the program:

```
% ex bigger.p
"bigger.p" 26 lines, 538 characters
```

```
:17
        x := d / i
:s'/'*
        x := d * i
:write
"bigger.p" 26 lines, 538 characters
:q
% pix bigger.p
Execution begins...
```



```
Execution terminated.

2550 statements executed in 0.16 seconds cpu time.
%
```

This appears to be the output we wanted. We could now save the output in a file if we wished by using the shell to redirect the output:

```
% px > graph
```

We can use *cat* (1) to see the contents of the file graph. We can also make a listing of the graph on the line printer without putting it into a file, e.g.

```
% px | lpr
Execution begins...
Execution terminated.

2550 statements executed in 0.15 seconds cpu time.
%
```

Note here that the statistics lines came out on our terminal. The statistics line comes out on the diagnostic output (unit 2.) There are two ways to get rid of the statistics line. We can redirect the statistics message to the printer using the syntax '| &' to the shell rather than '|', i.e.:

```
% px | & lpr
%
```

or we can translate the program with the **p** option disabled on the command line as we did above. This will disable all post-mortem dumping including the statistics line, thus:

```
% pi -p bigger.p
% px | lpr
%
```

This option also disables the statement limit which normally guards against infinite looping. You should not use it until your program is debugged. Also if **p** is specified and an error occurs, you will not get run time diagnostic information to help you determine what the problem is.

2.5. Formatting the program listing

It is possible to use special lines within the source text of a program to format the program listing. An empty line (one with no characters on it) corresponds to a 'space' macro in an assembler, leaving a completely blank line without a line number. A line containing only a control-l (form-feed) character will cause a page eject in the listing with the corresponding line number suppressed. This corresponds to an 'eject' pseudo-instruction. See also section 5.2 for details on the **n** and **i** options of *pi*.

2.6. Execution profiling

An execution profile consists of a structured listing of (all or part of) a program with information about the number of times each statement in the program was executed for a particular run of the program. These profiles can be used for several purposes. In a program which was abnormally terminated due to excessive looping or recursion or by a program fault, the counts can facilitate location of the error. Zero counts mark portions of the program which were not executed; during the early debugging stages they should prompt new test data or a re-examination of the program logic. The profile is perhaps most valuable, however, in drawing attention to the (typically small) portions of the program that dominate execution time. This information can be used for source level optimization.

An example

A prime number is a number which is divisible only by itself and the number one. The program *primes*, written by Niklaus Wirth, determines the first few prime numbers. In translating the program we have specified the **z** option to *pix*. This option causes the translator to generate counters and count instructions sufficient in number to determine the number of times each statement in the program was executed.† When execution of the program

†The counts are completely accurate only in the absence of runtime errors and nonlocal **goto** statements. This is not generally a problem, however, as in structured programs nonlocal **goto** statements occur infrequently, and counts are incorrect after abnormal termination only when the *upward look* described below to

completes, either normally or abnormally, this count data is written to the file *pmon.out* in the current directory.‡ It is then possible to prepare an execution profile by giving *pxp* the name of the file associated with this data, as was done in the following example.

```
% pix -l -z primes.p
Berkeley Pascal PI -- Version 3.1 (9/7/85)

Wed May  7 14:56 1986  primes.p

    1  program primes(output);
    2  const n = 50; n1 = 7; (*n1 = sqrt(n)*)
    3  var i,k,x,inc,lim,square,l: integer;
    4        prim: boolean;
    5        p,v: array[1..n1] of integer;
    6  begin
    7      write(2:6, 3:6); l := 2;
    8      x := 1; inc := 4; lim := 1; square := 9;
    9      for i := 3 to n do
   10      begin (*find next prime*)
   11          repeat x := x + inc; inc := 6-inc;
   12              if square <= x then
   13                  begin lim := lim+1;
   14                      v[lim] := square; square := sqr(p[lim+1])
   15                  end ;
   16              k := 2; prim := true;
   17              while prim and (k<lim) do
   18              begin k := k+1;
   19                  if v[k] < x then v[k] := v[k] + 2*p[k];
   20                  prim := x <> v[k]
   21              end
   22          until prim;
   23          if i <= n1 then p[i] := x;
   24          write(x:6); l := l+1;
   25          if l = 10 then
   26              begin writeln; l := 0
   27              end
   28      end ;
   29      writeln;
   30  end .
Execution begins...
     2     3     5     7    11    13    17    19    23    29
    31    37    41    43    47    53    59    61    67    71
    73    79    83    89    97   101   103   107   109   113
   127   131   137   139   149   151   157   163   167   173
   179   181   191   193   197   199   211   223   227   229

Execution terminated.

1404 statements executed in 0.08 seconds cpu time.
%
```

get a count passes a suspended call point.

‡*Pmon.out* has a name similar to *mon.out* the monitor file produced by the profiling facility of the C compiler *cc* (1). See *prof* (1) for a discussion of the C compiler profiling facilities.

Discussion

The header lines of the outputs of *pix* and *pxp* in this example indicate the version of the translator and execution profiler in use at the time this example was prepared. The time given with the file name (also on the header line) indicates the time of last modification of the program source file. This time serves to *version stamp* the input program. *Pxp* also indicates the time at which the profile data was gathered.

```
% pxp -z primes.p
Berkeley Pascal PXP -- Version 2.13 (4/2/84)

Wed May  7 14:56 1986  primes.p

Profiled Wed May  7 18:18 1986

     1          1. -----|program primes(output);
     2                  |const
     2                  |    n = 50;
     2                  |    nl = 7; (*nl = sqrt(n)*)
     3                  |var
     3                  |    i, k, x, inc, lim, square, l: integer;
     4                  |    prim: boolean;
     5                  |    p, v: array [1..nl] of integer;
     6                  |begin
     7                  |    write(2: 6, 3: 6);
     7                  |    l := 2;
     8                  |    x := 1;
     8                  |    inc := 4;
     8                  |    lim := 1;
     8                  |    square := 9;
     9                  |    for i := 3 to n do begin (*find next prime*)
     9               48. -----|    repeat
    11                   76. -----|    x := x + inc;
    11                            |    inc := 6 - inc;
    12                            |    if square <= x then begin
    13                         5. -----|    lim := lim + 1;
    14                            |    v[lim] := square;
    14                            |    square := sqr(p[lim + 1])
    14                       |    end;
    16                       |    k := 2;
    16                       |    prim := true;
    17                       |    while prim and (k < lim) do begin
    18                   157. -----|    k := k + 1;
    19                            |    if v[k] < x then
    19                        42. -----|    v[k] := v[k] + 2 * p[k];
    20                            |    prim := x <> v[k]
    20                       |    end
    20                       |until prim;
    23                  |    if i <= nl then
    23                5. -----|    p[i] := x;
    24                  |    write(x: 6);
    24                  |    l := l + 1;
    25                  |    if l = 10 then begin
    26                5. -----|    writeln;
    26                       |    l := 0
```

```
    26              |   end
    26          |   end;
    29          |   writeln
    29          |end.
%
```

To determine the number of times a statement was executed, one looks to the left of the statement and finds the corresponding vertical bar '|'. If this vertical bar is labelled with a count then that count gives the number of times the statement was executed. If the bar is not labelled, we look up in the listing to find the first '|' which directly above the original one which has a count and that is the answer. Thus, in our example, *k* was incremented 157 times on line 18, while the *write* procedure call on line 24 was executed 48 times as given by the count on the **repeat**.

More information on *pxp* can be found in its manual section *pxp* (1) and in sections 5.4, 5.5 and 5.10.

3. Error diagnostics

This section of the *User's Manual* discusses the error diagnostics of the programs *pi*, *pc* and *px*. *Pix* is a simple but useful program which invokes *pi* and *px* to do all the real processing. See its manual section *pix* (1) and section 5.2 below for more details. All the diagnostics given by *pi* will also be given by *pc*.

3.1. Translator syntax errors

A few comments on the general nature of the syntax errors usually made by Pascal programmers and the recovery mechanisms of the current translator may help in using the system.

Illegal characters

Characters such as '$', '!', and '@' are not part of the language Pascal. If they are found in the source program, and are not part of a constant string, a constant character, or a comment, they are considered to be 'illegal characters'. This can happen if you leave off an opening string quote '. Note that the character ', although used in English to quote strings, is not used to quote strings in Pascal. Most non-printing characters in your input are also illegal except in character constants and character strings. Except for the tab and form feed characters, which are used to ease formatting of the program, non-printing characters in the input file print as the character '?' so that they will show in your listing.

String errors

There is no character string of length 0 in Pascal. Consequently the input '' is not acceptable. Similarly, encountering an end-of-line after an opening string quote ' without encountering the matching closing quote yields the diagnostic "Unmatched ' for string". It is permissible to use the character '#' instead of ' to delimit character and constant strings for portability reasons. For this reason, a spuriously placed '#' sometimes causes the diagnostic about unbalanced quotes. Similarly, a '#' in column one is used when preparing programs which are to be kept in multiple files. See section 5.11 for details.

Comments in a comment, non-terminated comments

As we saw above, these errors are usually caused by leaving off a comment delimiter. You can convert parts of your program to comments without generating this diagnostic since there are two different kinds of comments – those delimited by '{' and '}', and those delimited by '(*' and '*)'. Thus consider:

```
{ This is a comment enclosing a piece of program
```

```
a := functioncall;      (* comment within comment *)
procedurecall;
lhs := rhs;         (* another comment *)
}
```

By using one kind of comment exclusively in your program you can use the other delimiters when you need to "comment out" parts of your program†. In this way you will also allow the translator to help by detecting statements accidentally placed within comments.

If a comment does not terminate before the end of the input file, the translator will point to the beginning of the comment, indicating that the comment is not terminated. In this case processing will terminate immediately. See the discussion of "QUIT" below.

Digits in numbers

This part of the language is a minor nuisance. Pascal requires digits in real numbers both before and after the decimal point. Thus the following statements, which look quite reasonable to FORTRAN users, generate diagnostics in Pascal:

```
Wed May  7 14:56 1986  digits.p:
    4  r := 0.;
e  -------------↑ ------ Digits required after decimal point
    5  r := .0;
e  -----------↑ ------ Digits required before decimal point
    6  r := 1.e10;
e  -------------↑ ------ Digits required after decimal point
    7  r := .05e-10;
e  -----------↑ ------ Digits required before decimal point
```

These same constructs are also illegal as input to the Pascal interpreter *px*.

Replacements, insertions, and deletions

When a syntax error is encountered in the input text, the parser invokes an error recovery procedure. This procedure examines the input text immediately after the point of error and considers a set of simple corrections to see whether they will allow the analysis to continue. These corrections involve replacing an input token with a different token, inserting a token, or replacing an input token with a different token. Most of these changes will not cause fatal syntax errors. The exception is the insertion of or replacement with a symbol such as an identifier or a number; in this case the recovery makes no attempt to determine *which* identifier or *what* number should be inserted, hence these are considered fatal syntax errors.

Consider the following example.

```
% pix -l synerr.p
Berkeley Pascal PI -- Version 3.1 (9/7/85)

Wed May  7 14:56 1986  synerr.p

    1  program syn(output);
    2  var i, j are integer;
e  ---------------↑--- Replaced identifier with a ':'
    3  begin
    4          for j :* 1 to 20 begin
```

†If you wish to transport your program, especially to the 6000-3.4 implementation, you should use the character sequence '(*' to delimit comments. For transportation over the *rcslink* to Pascal 6000-3.4, the character '#' should be used to delimit characters and constant strings.

```
e ------------------↑--- Replaced '*' with a '='
e ----------------------------↑--- Inserted keyword do
   5              write(j);
   6              i = 2 ** j;
e ---------------------↑--- Inserted ';'
E ------------------------------↑--- Inserted identifier
   7              writeln(i))
E ----------------------------------↑--- Deleted ')'
   8       end
   9 end.
%
```

The only surprise here may be that Pascal does not have an exponentiation operator, hence the complaint about '**'. This error illustrates that, if you assume that the language has a feature which it does not, the translator diagnostic may not indicate this, as the translator is unlikely to recognize the construct you supply.

Undefined or improper identifiers

If an identifier is encountered in the input but is undefined, the error recovery will replace it with an identifier of the appropriate class. Further references to this identifier will be summarized at the end of the containing **procedure** or **function** or at the end of the **program** if the reference occurred in the main program. Similarly, if an identifier is used in an inappropriate way, e.g. if a **type** identifier is used in an assignment statement, or if a simple variable is used where a **record** variable is required, a diagnostic will be produced and an identifier of the appropriate type inserted. Further incorrect references to this identifier will be flagged only if they involve incorrect use in a different way, with all incorrect uses being summarized in the same way as undefined variable uses are.

Expected symbols, malformed constructs

If none of the above mentioned corrections appear reasonable, the error recovery will examine the input to the left of the point of error to see if there is only one symbol which can follow this input. If this is the case, the recovery will print a diagnostic which indicates that the given symbol was 'Expected'.

In cases where none of these corrections resolve the problems in the input, the recovery may issue a diagnostic that indicates that the input is "malformed". If necessary, the translator may then skip forward in the input to a place where analysis can continue. This process may cause some errors in the text to be missed.

Consider the following example:

```
% pix -l synerr2.p
Berkeley Pascal PI -- Version 3.1 (9/7/85)

Wed May  7 14:56 1986  synerr2.p

   1 program synerr2(input,outpu);
   2 integer a(10)
E -----↑ ------ Malformed declaration
   3 begin
   4       read(b);
E ------------------↑ ------ Undefined variable
   5       for c := 1 to 10 do
E ----------------↑ ------ Undefined variable
   6             a(c) := b * c;
E ------------------↑ ------ Undefined procedure
```

```
E ------------------------↑ ------ Malformed statement
    7 end.
E 1 - File outpu listed in program statement but not declared
In program synerr2:
  E - a undefined on lines 6
  E - b undefined on line 4
  E - c undefined on line 5 6
Execution suppressed due to compilation errors
%
```

Here we misspelled *output* and gave a FORTRAN style variable declaration which the translator diagnosed as a 'Malformed declaration'. When, on line 6, we used '(' and ')' for subscripting (as in FORTRAN) rather than the '[' and ']' which are used in Pascal, the translator noted that *a* was not defined as a **procedure**. This occurred because **procedure** and **function** argument lists are delimited by parentheses in Pascal. As it is not permissible to assign to procedure calls the translator diagnosed a malformed statement at the point of assignment.

Expected and unexpected end-of-file, "QUIT"

If the translator finds a complete program, but there is more non-comment text in the input file, then it will indicate that an end-of-file was expected. This situation may occur after a bracketing error, or if too many **end**s are present in the input. The message may appear after the recovery says that it "Expected '.'" since '.' is the symbol that terminates a program.

If severe errors in the input prohibit further processing the translator may produce a diagnostic followed by "QUIT". One example of this was given above - a non-terminated comment; another example is a line which is longer than 160 characters. Consider also the following example.

```
% pix -l mism.p
Berkeley Pascal PI -- Version 3.1 (9/7/85)

Wed May  7 14:56 1986  mism.p

    1 program mismatch(output)
    2 begin
e -----↑ ------ Inserted ';'
    3       writeln('***');
    4       { The next line is the last line in the file }
    5       writeln
E ----------------------↑ ------ Malformed declaration
E ----------------------↑ ------ Unexpected end-of-file - QUIT
%
```

3.2. Translator semantic errors

The extremely large number of semantic diagnostic messages which the translator produces make it unreasonable to discuss each message or group of messages in detail. The messages are, however, very informative. We will here explain the typical formats and the terminology used in the error messages so that you will be able to make sense out of them. In any case in which a diagnostic is not completely comprehensible you can refer to the *User Manual* by Jensen and Wirth for examples.

Format of the error diagnostics

As we saw in the example program above, the error diagnostics from the Pascal translator include the number of a line in the text of the program as well as the text of the error

message. While this number is most often the line where the error occurred, it is occasionally the number of a line containing a bracketing keyword like **end** or **until**. In this case, the diagnostic may refer to the previous statement. This occurs because of the method the translator uses for sampling line numbers. The absence of a trailing ';' in the previous statement causes the line number corresponding to the **end** or **until**. to become associated with the statement. As Pascal is a free-format language, the line number associations can only be approximate and may seem arbitrary to some users. This is the only notable exception, however, to reasonable associations.

Incompatible types

Since Pascal is a strongly typed language, many semantic errors manifest themselves as type errors. These are called 'type clashes' by the translator. The types allowed for various operators in the language are summarized on page 108 of the Jensen-Wirth *User Manual*. It is important to know that the Pascal translator, in its diagnostics, distinguishes between the following type 'classes':

array	Boolean	char	file	integer
pointer	real	record	scalar	string

These words are plugged into a great number of error messages. Thus, if you tried to assign an *integer* value to a *char* variable you would receive a diagnostic like the following:

```
Wed May  7 14:56 1986  clash.p:
E 7 - Type clash: integer is incompatible with char
  ... Type of expression clashed with type of variable in assignment
```

In this case, one error produced a two line error message. If the same error occurs more than once, the same explanatory diagnostic will be given each time.

Scalar

The only class whose meaning is not self-explanatory is 'scalar'. Scalar has a precise meaning in the Jensen-Wirth *User Manual* where, in fact, it refers to *char*, *integer*, *real*, and *Boolean* types as well as the enumerated types. For the purposes of the Pascal translator, scalar in an error message refers to a user-defined, enumerated type, such as *ops* in the example above or *color* in

```
type color = (red, green, blue)
```

For integers, the more explicit denotation *integer* is used. Although it would be correct, in the context of the *User Manual* to refer to an integer variable as a *scalar* variable *pi* prefers the more specific identification.

Function and procedure type errors

For built-in procedures and functions, two kinds of errors occur. If the routines are called with the wrong number of arguments a message similar to:

```
Wed May  7 14:56 1986  sin1.p:
E 12 - sin takes exactly one argument
```

is given. If the type of the argument is wrong, a message like

```
Wed May  7 14:56 1986  sin2.p:
E 12 - sin's argument must be integer or real, not char
```

is produced. A few functions and procedures implemented in Pascal 6000-3.4 are diagnosed as unimplemented in Berkeley Pascal, notably those related to **segmented** files.

Can't read and write scalars, etc.

The messages which state that scalar (user-defined) types cannot be written to and from files are often mysterious. It is in fact the case that if you define

type color = (red, green, blue)

"standard" Pascal does not associate these constants with the strings 'red', 'green', and 'blue' in any way. An extension has been added which allows enumerated types to be read and written, however if the program is to be portable, you will have to write your own routines to perform these functions. Standard Pascal only allows the reading of characters, integers and real numbers from text files. You cannot read strings or Booleans. It is possible to make a

file of color

but the representation is binary rather than string.

Expression diagnostics

The diagnostics for semantically ill-formed expressions are very explicit. Consider this sample translation:

```
% pi -l expr.p
Berkeley Pascal PI -- Version 3.1 (9/7/85)

Wed May  7 14:56 1986  expr.p

    1  program x(output);
    2  var
    3          a: set of char;
    4          b: Boolean;
    5          c: (red, green, blue);
    6          p: ↑ integer;
    7          A: alfa;
    8          B: packed array [1..5] of char;
    9  begin
   10          b := true;
   11          c := red;
   12          new(p);
   13          a := [];
   14          A := 'Hello, yellow';
   15          b := a and b;
   16          a := a * 3;
   17          if input < 2 then writeln('boo');
   18          if p <= 2 then writeln('sure nuff');
   19          if A = B then writeln('same');
   20          if c = true then writeln('hue''s and color''s')
   21  end.
E 14 - Constant string too long
E 15 - Left operand of and must be Boolean, not set
E 16 - Cannot mix sets with integers and reals as operands of *
E 17 - files may not participate in comparisons
E 18 - pointers and integers cannot be compared - operator was <=
E 19 - Strings not same length in = comparison
E 20 - scalars and Booleans cannot be compared - operator was =
e 21 - Input is used but not defined in the program statement
In program x:
```

```
  w - constant green is never used
  w - constant blue is never used
  w - variable B is used but never set
%
```

This example is admittedly far-fetched, but illustrates that the error messages are sufficiently clear to allow easy determination of the problem in the expressions.

Type equivalence

Several diagnostics produced by the Pascal translator complain about 'non-equivalent types'. In general, Berkeley Pascal considers variables to have the same type only if they were declared with the same constructed type or with the same type identifier. Thus, the variables *x* and *y* declared as

```
var
   x: ↑ integer;
   y: ↑ integer;
```

do not have the same type. The assignment

```
x := y
```

thus produces the diagnostics:

```
Wed May  7 14:56 1986  typequ.p:
E 7 - Type clash: non-identical pointer types
  ... Type of expression clashed with type of variable in assignment
```

Thus it is always necessary to declare a type such as

```
type intptr = ↑ integer;
```

and use it to declare

```
var x: intptr; y: intptr;
```

Note that if we had initially declared

```
var x, y: ↑ integer;
```

then the assignment statement would have worked. The statement

```
x↑ := y↑
```

is allowed in either case. Since the parameter to a **procedure** or **function** must be declared with a type identifier rather than a constructed type, it is always necessary, in practice, to declare any type which will be used in this way.

Unreachable statements

Berkeley Pascal flags unreachable statements. Such statements usually correspond to errors in the program logic. Note that a statement is considered to be reachable if there is a potential path of control, even if it can never be taken. Thus, no diagnostic is produced for the statement:

```
if false then
   writeln('impossible!')
```

Goto's into structured statements

The translator detects and complains about **goto** statements which transfer control into structured statements (**for**, **while**, etc.) It does not allow such jumps, nor does it allow branching from the **then** part of an **if** statement into the **else** part. Such checks are made only within the body of a single procedure or function.

Unused variables, never set variables

Although *pi* always clears variables to 0 at **procedure** and **function** entry, *pc* does not unless runtime checking is enabled using the **C** option. It is **not** good programming practice to rely on this initialization. To discourage this practice, and to help detect errors in program logic, *pi* flags as a 'w' warning error:

1) Use of a variable which is never assigned a value.
2) A variable which is declared but never used, distinguishing between those variables for which values are computed but which are never used, and those completely unused.

In fact, these diagnostics are applied to all declared items. Thus a **const** or a **procedure** which is declared but never used is flagged. The **w** option of *pi* may be used to suppress these warnings; see sections 5.1 and 5.2.

3.3. Translator panics, i/o errors

Panics

One class of error which rarely occurs, but which causes termination of all processing when it does is a panic. A panic indicates a translator-detected internal inconsistency. A typical panic message is:

```
snark (rvalue) line=110 yyline=109
Snark in pi
```

If you receive such a message, the translation will be quickly and perhaps ungracefully terminated. You should contact a teaching assistant or a member of the system staff, after saving a copy of your program for later inspection. If you were making changes to an existing program when the problem occurred, you may be able to work around the problem by ascertaining which change caused the *snark* and making a different change or correcting an error in the program. A small number of panics are possible in *px*. All panics should be reported to a teaching assistant or systems staff so that they can be fixed.

Out of memory

The only other error which will abort translation when no errors are detected is running out of memory. All tables in the translator, with the exception of the parse stack, are dynamically allocated, and can grow to take up the full available process space of 64000 bytes on the PDP-11. On the VAX-11, table sizes are extremely generous and very large (25000) line programs have been easily accommodated. For the PDP-11, it is generally true that the size of the largest translatable program is directly related to **procedure** and **function** size. A number of non-trivial Pascal programs, including some with more than 2000 lines and 2500 statements have been translated and interpreted using Berkeley Pascal on PDP-11's. Notable among these are the Pascal-S interpreter, a large set of programs for automated generation of code generators, and a general context-free parsing program which has been used to parse sentences with a grammar for a superset of English. In general, very large programs should be translated using *pc* and the separate compilation facility.

If you receive an out of space message from the translator during translation of a large **procedure** or **function** or one containing a large number of string constants you may yet be able to translate your program if you break this one **procedure** or **function** into several

routines.

I/O errors

Other errors which you may encounter when running *pi* relate to input-output. If *pi* cannot open the file you specify, or if the file is empty, you will be so informed.

3.4. Run-time errors

We saw, in our second example, a run-time error. We here give the general description of run-time errors. The more unusual interpreter error messages are explained briefly in the manual section for *px* (1).

Start-up errors

These errors occur when the object file to be executed is not available or appropriate. Typical errors here are caused by the specified object file not existing, not being a Pascal object, or being inaccessible to the user.

Program execution errors

These errors occur when the program interacts with the Pascal runtime environment in an inappropriate way. Typical errors are values or subscripts out of range, bad arguments to built-in functions, exceeding the statement limit because of an infinite loop, or running out of memory‡. The interpreter will produce a backtrace after the error occurs, showing all the active routine calls, unless the **p** option was disabled when the program was translated. Unfortunately, no variable values are given and no way of extracting them is available.*

As an example of such an error, assume that we have accidentally declared the constant *n1* to be 6, instead of 7 on line 2 of the program primes as given in section 2.6 above. If we run this program we get the following response.

```
% pix primes.p
Execution begins...
   2    3    5    7   11   13   17   19   23   29
  31   37   41   43   47   53   59   61   67   71
  73   79   83   89   97  101  103  107  109  113
 127  131  137  139  149  151  157  163  167
Subscript value of 7 is out of range

        Error in "primes"+8 near line 14.
Execution terminated abnormally.

941 statements executed in 0.07 seconds cpu time.
%
```

Here the interpreter indicates that the program terminated abnormally due to a subscript out of range near line 14, which is eight lines into the body of the program primes.

Interrupts

If the program is interrupted while executing and the **p** option was not specified, then a backtrace will be printed.† The file *pmon.out* of profile information will be written if the

‡The checks for running out of memory are not foolproof and there is a chance that the interpreter will fault, producing a core image when it runs out of memory. This situation occurs very rarely.

* On the VAX-11, each variable is restricted to allocate at most 65000 bytes of storage (this is a PDP-11ism that has survived to the VAX.)

†Occasionally, the Pascal system will be in an inconsistent state when this occurs, e.g. when an interrupt terminates a **procedure** or **function** entry or exit. In this case, the backtrace will only contain the current

program was translated with the z option enabled to *pi* or *pix*.

I/O interaction errors

The final class of interpreter errors results from inappropriate interactions with files, including the user's terminal. Included here are bad formats for integer and real numbers (such as no digits after the decimal point) when reading.

4. Input/output

This section describes features of the Pascal input/output environment, with special consideration of the features peculiar to an interactive implementation.

4.1. Introduction

Our first sample programs, in section 2, used the file *output*. We gave examples there of redirecting the output to a file and to the line printer using the shell. Similarly, we can read the input from a file or another program. Consider the following Pascal program which is similar to the program *cat* (1).

```
% pix -l kat.p <primes
Berkeley Pascal PI -- Version 3.1 (9/7/85)

Wed May  7 14:56 1986  kat.p

    1   program kat(input, output);
    2   var
    3       ch: char;
    4   begin
    5       while not eof do begin
    6           while not eoln do begin
    7               read(ch);
    8               write(ch)
    9           end;
   10           readln;
   11           writeln
   12       end
   13   end { kat }.
Execution begins...
    2     3     5     7    11    13    17    19    23    29
   31    37    41    43    47    53    59    61    67    71
   73    79    83    89    97   101   103   107   109   113
  127   131   137   139   149   151   157   163   167   173
  179   181   191   193   197   199   211   223   227   229

Execution terminated.

925 statements executed in 0.06 seconds cpu time.
%
```

Here we have used the shell's syntax to redirect the program input from a file in *primes* in which we had placed the output of our prime number program of section 2.6. It is also possible to 'pipe' input to this program much as we piped input to the line printer daemon *lpr* (1) before. Thus, the same output as above would be produced by

line. A reverse call order list of procedures will not be given.

```
% cat primes | pix -l kat.p
```

All of these examples use the shell to control the input and output from files. One very simple way to associate Pascal files with named UNIX files is to place the file name in the **program** statement. For example, suppose we have previously created the file *data*. We then use it as input to another version of a listing program.

```
% cat data
line one.
line two.
line three is the end.
% pix -l copydata.p
Berkeley Pascal PI -- Version 3.1 (9/7/85)

Wed May  7 14:56 1986  copydata.p

    1  program copydata(data, output);
    2  var
    3          ch: char;
    4          data: text;
    5  begin
    6          reset(data);
    7          while not eof(data) do begin
    8                  while not eoln(data) do begin
    9                          read(data, ch);
   10                          write(ch)
   11                  end;
   12                  readln(data);
   13                  writeln
   14          end
   15  end { copydata }.
Execution begins...
line one.
line two.
line three is the end.
Execution terminated.

134 statements executed in 0.02 seconds cpu time.
%
```

By mentioning the file *data* in the **program** statement, we have indicated that we wish it to correspond to the UNIX file *data*. Then, when we 'reset(data)', the Pascal system opens our file 'data' for reading. More sophisticated, but less portable, examples of using UNIX files will be given in sections 4.5 and 4.6. There is a portability problem even with this simple example. Some Pascal systems attach meaning to the ordering of the file in the **program** statement file list. Berkeley Pascal does not do so.

4.2. Eof and eoln

An extremely common problem encountered by new users of Pascal, especially in the interactive environment offered by UNIX, relates to the definitions of *eof* and *eoln*. These functions are supposed to be defined at the beginning of execution of a Pascal program, indicating whether the input device is at the end of a line or the end of a file. Setting *eof* or *eoln* actually corresponds to an implicit read in which the input is inspected, but no input is "used up". In fact, there is no way the system can know whether the input is at the end-of-file or the end-of-line unless it attempts to read a line from it. If the input is from a previously

created file, then this reading can take place without run-time action by the user. However, if the input is from a terminal, then the input is what the user types.† If the system were to do an initial read automatically at the beginning of program execution, and if the input were a terminal, the user would have to type some input before execution could begin. This would make it impossible for the program to begin by prompting for input or printing a herald.

Berkeley Pascal has been designed so that an initial read is not necessary. At any given time, the Pascal system may or may not know whether the end-of-file or end-of-line conditions are true. Thus, internally, these functions can have three values – true, false, and "I don't know yet; if you ask me I'll have to find out". All files remain in this last, indeterminate state until the Pascal program requires a value for *eof* or *eoln* either explicitly or implicitly, e.g. in a call to *read*. The important point to note here is that if you force the Pascal system to determine whether the input is at the end-of-file or the end-of-line, it will be necessary for it to attempt to read from the input.

Thus consider the following example code

```
while not eof do begin
    write('number, please? ');
    read(i);
    writeln('that was a ', i: 2)
end
```

At first glance, this may be appear to be a correct program for requesting, reading and echoing numbers. Notice, however, that the **while** loop asks whether *eof* is true *before* the request is printed. This will force the Pascal system to decide whether the input is at the end-of-file. The Pascal system will give no messages; it will simply wait for the user to type a line. By producing the desired prompting before testing *eof,* the following code avoids this problem:

```
write('number, please ?');
while not eof do begin
    read(i);
    writeln('that was a ', i:2);
    write('number, please ?')
end
```

The user must still type a line before the **while** test is completed, but the prompt will ask for it. This example, however, is still not correct. To understand why, it is first necessary to know, as we will discuss below, that there is a blank character at the end of each line in a Pascal text file. The *read* procedure, when reading integers or real numbers, is defined so that, if there are only blanks left in the file, it will return a zero value and set the end-of-file condition. If, however, there is a number remaining in the file, the end-of-file condition will not be set even if it is the last number, as *read* never reads the blanks after the number, and there is always at least one blank. Thus the modified code will still put out a spurious

```
that was a 0
```

at the end of a session with it when the end-of-file is reached. The simplest way to correct the problem in this example is to use the procedure *readln* instead of *read* here. In general, unless we test the end-of-file condition both before and after calls to *read* or *readln*, there will be inputs for which our program will attempt to read past end-of-file.

†It is not possible to determine whether the input is a terminal, as the input may appear to be a file but actually be a *pipe,* the output of a program which is reading from the terminal.

4.3. More about eoln

To have a good understanding of when *eoln* will be true it is necessary to know that in any file there is a special character indicating end-of-line, and that, in effect, the Pascal system always reads one character ahead of the Pascal *read* commands.† For instance, in response to 'read(ch)', the system sets *ch* to the current input character and gets the next input character. If the current input character is the last character of the line, then the next input character from the file is the new-line character, the normal UNIX line separator. When the read routine gets the new-line character, it replaces that character by a blank (causing every line to end with a blank) and sets *eoln* to true. *Eoln* will be true as soon as we read the last character of the line and **before** we read the blank character corresponding to the end of line. Thus it is almost always a mistake to write a program which deals with input in the following way:

```
read(ch);
if eoln then
    Done with line
else
    Normal processing
```

as this will almost surely have the effect of ignoring the last character in the line. The 'read(ch)' belongs as part of the normal processing.

Given this framework, it is not hard to explain the function of a *readln* call, which is defined as:

```
while not eoln do
    get(input);
get(input);
```

This advances the file until the blank corresponding to the end-of-line is the current input symbol and then discards this blank. The next character available from *read* will therefore be the first character of the next line, if one exists.

4.4. Output buffering

A final point about Pascal input-output must be noted here. This concerns the buffering of the file *output*. It is extremely inefficient for the Pascal system to send each character to the user's terminal as the program generates it for output; even less efficient if the output is the input of another program such as the line printer daemon *lpr* (1). To gain efficiency, the Pascal system "buffers" the output characters (i.e. it saves them in memory until the buffer is full and then emits the entire buffer in one system interaction.) However, to allow interactive prompting to work as in the example given above, this prompt must be printed before the Pascal system waits for a response. For this reason, Pascal normally prints all the output which has been generated for the file *output* whenever

1) A *writeln* occurs, or
2) The program reads from the terminal, or
3) The procedure *message* or *flush* is called.

Thus, in the code sequence

```
for i := 1 to 5 do begin
    write(i: 2);
    Compute a lot with no output
end;
writeln
```

†In Pascal terms, 'read(ch)' corresponds to 'ch := input†; get(input)'

the output integers will not print until the *writeln* occurs. The delay can be somewhat disconcerting, and you should be aware that it will occur. By setting the **b** option to 0 before the **program** statement by inserting a comment of the form

```
(*$b0*)
```

we can cause *output* to be completely unbuffered, with a corresponding horrendous degradation in program efficiency. Option control in comments is discussed in section 5.

4.5. Files, reset, and rewrite

It is possible to use extended forms of the built-in functions *reset* and *rewrite* to get more general associations of UNIX file names with Pascal file variables. When a file other than *input* or *output* is to be read or written, then the reading or writing must be preceded by a *reset* or *rewrite* call. In general, if the Pascal file variable has never been used before, there will be no UNIX filename associated with it. As we saw in section 2.9, by mentioning the file in the **program** statement, we could cause a UNIX file with the same name as the Pascal variable to be associated with it. If we do not mention a file in the **program** statement and use it for the first time with the statement

```
reset(f)
```

or

```
rewrite(f)
```

then the Pascal system will generate a temporary name of the form 'tmp.x' for some character 'x', and associate this UNIX file name name with the Pascal file. The first such generated name will be 'tmp.1' and the names continue by incrementing their last character through the ASCII set. The advantage of using such temporary files is that they are automatically *removed* by the Pascal system as soon as they become inaccessible. They are not removed, however, if a runtime error causes termination while they are in scope.

To cause a particular UNIX pathname to be associated with a Pascal file variable we can give that name in the *reset* or *rewrite* call, e.g. we could have associated the Pascal file *data* with the file 'primes' in our example in section 3.1 by doing:

```
reset(data, 'primes')
```

instead of a simple

```
reset(data)
```

In this case it is not essential to mention 'data' in the program statement, but it is still a good idea because is serves as an aid to program documentation. The second parameter to *reset* and *rewrite* may be any string value, including a variable. Thus the names of UNIX files to be associated with Pascal file variables can be read in at run time. Full details on file name/file variable associations are given in section A.3.

4.6. Argc and argv

Each UNIX process receives a variable length sequence of arguments each of which is a variable length character string. The built-in function *argc* and the built-in procedure *argv* can be used to access and process these arguments. The value of the function *argc* is the number of arguments to the process. By convention, the arguments are treated as an array, and indexed from 0 to *argc*−1, with the zeroth argument being the name of the program being executed. The rest of the arguments are those passed to the command on the command line. Thus, the command

```
% obj /etc/motd /usr/dict/words hello
```

will invoke the program in the file *obj* with *argc* having a value of 4. The zeroth element accessed by *argv* will be 'obj', the first '/etc/motd', etc.

Pascal does not provide variable size arrays, nor does it allow character strings of varying length. For this reason, *argv* is a procedure and has the syntax

```
argv(i, a)
```

where *i* is an integer and *a* is a string variable. This procedure call assigns the (possibly truncated or blank padded) *i* 'th argument of the current process to the string variable *a*. The file manipulation routines *reset* and *rewrite* will strip trailing blanks from their optional second arguments so that this blank padding is not a problem in the usual case where the arguments are file names.

We are now ready to give a Berkeley Pascal program 'kat', based on that given in section 3.1 above, which can be used with the same syntax as the UNIX system program *cat* (1).

```
% cat kat.p
program kat(input, output);
var
    ch: char;
    i: integer;
    name: packed array [1..100] of char;
begin
    i := 1;
    repeat
        if i < argc then begin
            argv(i, name);
            reset(input, name);
            i := i + 1
        end;
        while not eof do begin
            while not eoln do begin
                read(ch);
                write(ch)
            end;
            readln;
            writeln
        end
    until i >= argc
end { kat }.
%
```

Note that the *reset* call to the file *input* here, which is necessary for a clear program, may be disallowed on other systems. As this program deals mostly with *argc* and *argv* and UNIX system dependent considerations, portability is of little concern.

If this program is in the file 'kat.p', then we can do

```
% pi kat.p
% mv obj kat
% kat primes
    2    3    5    7   11   13   17   19   23   29
   31   37   41   43   47   53   59   61   67   71
   73   79   83   89   97  101  103  107  109  113
  127  131  137  139  149  151  157  163  167  173
```

```
179   181   191   193   197   199   211   223   227   229

930 statements executed in 0.06 seconds cpu time.
% kat
This is a line of text.
This is a line of text.
The next line contains only an end-of-file (an invisible control-d!)
The next line contains only an end-of-file (an invisible control-d!)

287 statements executed in 0.02 seconds cpu time.
%
```

Thus we see that, if it is given arguments, 'kat' will, like *cat,* copy each one in turn. If no arguments are given, it copies from the standard input. Thus it will work as it did before, with

```
% kat < primes
```

now equivalent to

```
% kat primes
```

although the mechanisms are quite different in the two cases. Note that if 'kat' is given a bad file name, for example:

```
% kat xxxxqqq

Could not open xxxxqqq: No such file or directory

    Error in "kat"+5 near line 11.

4 statements executed in 0.00 seconds cpu time.
%
```

it will give a diagnostic and a post-mortem control flow backtrace for debugging. If we were going to use 'kat', we might want to translate it differently, e.g.:

```
% pi -pb kat.p
% mv obj kat
```

Here we have disabled the post-mortem statistics printing, so as not to get the statistics or the full traceback on error. The **b** option will cause the system to block buffer the input/output so that the program will run more efficiently on large files. We could have also specified the t option to turn off runtime tests if that was felt to be a speed hindrance to the program. Thus we can try the last examples again:

```
% kat xxxxqqq

Could not open xxxxqqq: No such file or directory

    Error in "kat"
% kat primes
  2    3    5    7   11   13   17   19   23   29
 31   37   41   43   47   53   59   61   67   71
 73   79   83   89   97  101  103  107  109  113
127  131  137  139  149  151  157  163  167  173
```

PS1:4

```
179   181   191   193   197   199   211   223   227   229
%
```

The interested reader may wish to try writing a program which accepts command line arguments like *pi* does, using *argc* and *argv* to process them.

5. Details on the components of the system

5.1. Options

The programs *pi*, *pc*, and *pxp* take a number of options.† There is a standard UNIX convention for passing options to programs on the command line, and this convention is followed by the Berkeley Pascal system programs. As we saw in the examples above, option related arguments consisted of the character '–' followed by a single character option name.

Except for the **b** option which takes a single digit value, each option may be set on (enabled) or off (disabled.) When an on/off valued option appears on the command line of *pi* or it inverts the default setting of that option. Thus

```
% pi -l foo.p
```

enables the listing option **l**, since it defaults off, while

```
% pi -t foo.p
```

disables the run time tests option **t**, since it defaults on.

In additon to inverting the default settings of *pi* options on the command line, it is also possible to control the *pi* options within the body of the program by using comments of a special form illustrated by

```
{$l-}
```

Here we see that the opening comment delimiter (which could also be a '(*') is immediately followed by the character '$'. After this '$', which signals the start of the option list, we can place a sequence of letters and option controls, separated by ',' characters‡. The most basic actions for options are to set them, thus

```
{$l+ Enable listing}
```

or to clear them

```
{$t-,p- No run-time tests, no post mortem analysis}
```

Notice that '+' always enables an option and '–' always disables it, no matter what the default is. Thus '–' has a different meaning in an option comment than it has on the command line. As shown in the examples, normal comment text may follow the option list.

†As *pix* uses *pi* to translate Pascal programs, it takes the options of *pi* also. We refer to them here, however, as *pi* options.

‡This format was chosen because it is used by Pascal 6000-3.4. In general the options common to both implementations are controlled in the same way so that comment control in options is mostly portable. It is recommended, however, that only one control be put per comment for maximum portability, as the Pascal 6000-3.4 implementation will ignore controls after the first one which it does not recognize.

5.2. Options common to Pi, Pc, and Pix

The following options are common to both the compiler and the interpreter. With each option we give its default setting, the setting it would have if it appeared on the command line, and a sample command using the option. Most options are on/off valued, with the **b** option taking a single digit value.

Buffering of the file output – b

The **b** option controls the buffering of the file *output*. The default is line buffering, with flushing at each reference to the file *input* and under certain other circumstances detailed in section 5 below. Mentioning **b** on the command line, e.g.

```
% pi -b assembler.p
```

causes standard output to be block buffered, where a block is some system-defined number of characters. The **b** option may also be controlled in comments. It, unique among the Berkeley Pascal options, takes a single digit value rather than an on or off setting. A value of 0, e.g.

```
{$b0}
```

causes the file *output* to be unbuffered. Any value 2 or greater causes block buffering and is equivalent to the flag on the command line. The option control comment setting **b** must precede the **program** statement.

Include file listing – i

The **i** option takes the name of an **include** file, **procedure** or **function** name and causes it to be listed while translating†. Typical uses would be

```
% pix -i scanner.i compiler.p
```

to make a listing of the routines in the file scanner.i, and

```
% pix -i scanner compiler.p
```

to make a listing of only the routine *scanner*. This option is especially useful for conservation-minded programmers making partial program listings.

Make a listing – l

The **l** option enables a listing of the program. The **l** option defaults off. When specified on the command line, it causes a header line identifying the version of the translator in use and a line giving the modification time of the file being translated to appear before the actual program listing. The **l** option is pushed and popped by the **i** option at appropriate points in the program.

Standard Pascal only – s

The **s** option causes many of the features of the UNIX implementation which are not found in standard Pascal to be diagnosed as 's' warning errors. This option defaults off and is enabled when mentioned on the command line. Some of the features which are diagnosed are: non-standard **procedures** and **functions**, extensions to the **procedure** *write*, and the padding of constant strings with blanks. In addition, all letters are mapped to lower case except in strings and characters so that the case of keywords and identifiers is effectively ignored. The **s** option is most useful when a program is to be transported, thus

†**Include** files are discussed in section 5.9.

```
% pi -s isitstd.p
```

will produce warnings unless the program meets the standard.

Runtime tests - t and C

These options control the generation of tests that subrange variable values are within bounds at run time. *pi* defaults to generating tests and uses the option **t** to disable them. *pc* defaults to not generating tests, and uses the option **C** to enable them. Disabling runtime tests also causes **assert** statements to be treated as comments.‡

Suppress warning diagnostics - w

The **w** option, which defaults on, allows the translator to print a number of warnings about inconsistencies it finds in the input program. Turning this option off with a comment of the form

```
{$w-}
```

or on the command line

```
% pi -w tryme.p
```

suppresses these usually useful diagnostics.

Generate counters for a pxp execution profile - z

The **z** option, which defaults off, enables the production of execution profiles. By specifying **z** on the command line, i.e.

```
% pi -z foo.p
```

or by enabling it in a comment before the **program** statement causes *pi* and *pc* to insert operations in the interpreter code to count the number of times each statement was executed. An example of using *pxp* was given in section 2.6; its options are described in section 5.6. Note that the **z** option cannot be used on separately compiled programs.

5.3. Options available in Pi

Post-mortem dump - p

The **p** option defaults on, and causes the runtime system to initiate a post-mortem backtrace when an error occurs. It also cause *px* to count statements in the executing program, enforcing a statement limit to prevent infinite loops. Specifying **p** on the command line disables these checks and the ability to give this post-mortem analysis. It does make smaller and faster programs, however. It is also possible to control the **p** option in comments. To prevent the post-mortem backtrace on error, **p** must be off at the end of the **program** statement. Thus, the Pascal cross-reference program was translated with

```
% pi -pbt pxref.p
```

5.4. Options available in Px

The first argument to *px* is the name of the file containing the program to be interpreted. If no arguments are given, then the file *obj* is executed. If more arguments are given, they are available to the Pascal program by using the built-ins *argc* and *argv* as described in

‡See section A.1 for a description of **assert** statements.

section 4.6.

Px may also be invoked automatically. In this case, whenever a Pascal object file name is given as a command, the command will be executed with *px* prepended to it; that is

```
% obj primes
```

will be converted to read

```
% px obj primes
```

5.5. Options available in Pc

Generate assembly language – S

The program is compiled and the assembly language output is left in file appended .s. Thus

```
% pc -S foo.p
```

creates a file *foo.s*. No executable file is created.

Symbolic Debugger Information – g

The **g** option causes the compiler to generate information needed by *sdb*(1) the symbolic debugger. For a complete description of *sdb* see Volume 2c of the UNIX Reference Manual.

Redirect the output file – o

The *name* argument after the **-o** is used as the name of the output file instead of *a.out*. Its typical use is to name the compiled program using the root of the file name. Thus:

```
% pc -o myprog myprog.p
```

causes the compiled program to be called *myprog*.

Generate counters for a *prof* execution profile – p

The compiler produces code which counts the number of times each routine is called. The profiling is based on a periodic sample taken by the system rather than by inline counters used by *pxp*. This results in less degradation in execution, at somewhat of a loss in accuracy. See *prof*(1) for a more complete description.

Run the object code optimizer – O

The output of the compiler is run through the object code optimizer. This provides an increase in compile time in exchange for a decrease in compiled code size and execution time.

5.6. Options available in Pxp

Pxp takes, on its command line, a list of options followed by the program file name, which must end in '.p' as it must for *pi*, *pc*, and *pix*. *Pxp* will produce an execution profile if any of the **z**, **t** or **c** options is specified on the command line. If none of these options is specified, then *pxp* functions as a program reformatter.

It is important to note that only the **z** and **w** options of *pxp*, which are common to *pi*, *pc*, and *pxp* can be controlled in comments. All other options must be specified on the command line to have any effect.

The following options are relevant to profiling with *pxp*:

Include the bodies of all routines in the profile – a

Pxp normally suppresses printing the bodies of routines which were never executed, to make the profile more compact. This option forces all routine bodies to be printed.

Suppress declaration parts from a profile – d

Normally a profile includes declaration parts. Specifying **d** on the command line suppresses declaration parts.

Eliminate include directives – e

Normally, *pxp* preserves **include** directives to the output when reformatting a program, as though they were comments. Specifying **–e** causes the contents of the specified files to be reformatted into the output stream instead. This is an easy way to eliminate **include** directives, e.g. before transporting a program.

Fully parenthesize expressions – f

Normally *pxp* prints expressions with the minimal parenthesization necessary to preserve the structure of the input. This option causes *pxp* to fully parenthesize expressions. Thus the statement which prints as

```
d := a + b mod c / e
```

with minimal parenthesization, the default, will print as

```
d := a + ((b mod c) / e)
```

with the **f** option specified on the command line.

Left justify all procedures and functions – j

Normally, each **procedure** and **function** body is indented to reflect its static nesting depth. This option prevents this nesting and can be used if the indented output would be too wide.

Print a table summarizing procedure and function calls – t

The **t** option causes *pxp* to print a table summarizing the number of calls to each **procedure** and **function** in the program. It may be specified in combination with the **z** option, or separately.

Enable and control the profile – z

The **z** profile option is very similar to the **i** listing control option of *pi*. If **z** is specified on the command line, then all arguments up to the source file argument which ends in '.p' are taken to be the names of **procedures** and **functions** or **include** files which are to be profiled. If this list is null, then the whole file is to be profiled. A typical command for extracting a profile of part of a large program would be

```
% pxp -z test parser.i compiler.p
```

This specifies that profiles of the routines in the file *parser.i* and the routine *test* are to be made.

5.7. Formatting programs using pxp

The program *pxp* can be used to reformat programs, by using a command of the form

```
% pxp dirty.p > clean.p
```

Note that since the shell creates the output file 'clean.p' before *pxp* executes, so 'clean.p' and 'dirty.p' must not be the same file.

Pxp automatically paragraphs the program, performing housekeeping chores such as comment alignment, and treating blank lines, lines containing exactly one blank and lines containing only a form-feed character as though they were comments, preserving their vertical spacing effect in the output. *Pxp* distinguishes between four kinds of comments:

1) Left marginal comments, which begin in the first column of the input line and are placed in the first column of an output line.
2) Aligned comments, which are preceded by no input tokens on the input line. These are aligned in the output with the running program text.
3) Trailing comments, which are preceded in the input line by a token with no more than two spaces separating the token from the comment.
4) Right marginal comments, which are preceded in the input line by a token from which they are separated by at least three spaces or a tab. These are aligned down the right margin of the output, currently to the first tab stop after the 40th column from the current "left margin".

Consider the following program.


```
% cat comments.p
{ This is a left marginal comment. }
program hello(output);
var i : integer; {This is a trailing comment}
j : integer;    {This is a right marginal comment}
k : array [ 1..10] of array [1..10] of integer; {Marginal, but past the margin}
{
  An aligned, multi-line comment
  which explains what this program is
  all about
}
begin
i := 1; {Trailing i comment}
{A left marginal comment}
 {An aligned comment}
j := 1;        {Right marginal comment}
k[1] := 1;
writeln(i, j, k[1])
end.
```

When formatted by *pxp* the following output is produced.

```
% pxp comments.p
{ This is a left marginal comment. }

program hello(output);
var
    i: integer; {This is a trailing comment}
    j: integer;                                  {This is a right marginal comment}
    k: array [1..10] of array [1..10] of integer;  {Marginal, but past the margin}
{
  An aligned, multi-line comment
  which explains what this program is
  all about
}
```

```
begin
   i := 1; {Trailing i comment}
{A left marginal comment}
   {An aligned comment}
   j := 1;                                   {Right marginal comment}
   k[1] := 1;
   writeln(i, j, k[1])
end.
%
```

The following formatting related options are currently available in *pxp*. The options **f** and **j** described in the previous section may also be of interest.

Strip comments -s

The s option causes *pxp* to remove all comments from the input text.

Underline keywords - _

A command line argument of the form -_ as in

```
% pxp -_ dirty.p
```

can be used to cause *pxp* to underline all keywords in the output for enhanced readability.

Specify indenting unit - [23456789]

The normal unit which *pxp* uses to indent a structure statement level is 4 spaces. By giving an argument of the form *-d* with *d* a digit, $2 \leq d \leq 9$ you can specify that *d* spaces are to be used per level instead.

5.8. Pxref

The cross-reference program *pxref* may be used to make cross-referenced listings of Pascal programs. To produce a cross-reference of the program in the file 'foo.p' one can execute the command:

```
% pxref foo.p
```

The cross-reference is, unfortunately, not block structured. Full details on *pxref* are given in its manual section *pxref* (1).

5.9. Multi-file programs

A text inclusion facility is available with Berkeley Pascal. This facility allows the interpolation of source text from other files into the source stream of the translator. It can be used to divide large programs into more manageable pieces for ease in editing, listing, and maintenance.

The **include** facility is based on that of the UNIX C compiler. To trigger it you can place the character '#' in the first portion of a line and then, after an arbitrary number of blanks or tabs, the word 'include' followed by a filename enclosed in single '' or double "" quotation marks. The file name may be followed by a semicolon ';' if you wish to treat this as a pseudo-Pascal statement. The filenames of included files must end in '.i'. An example of the use of included files in a main program would be:

```
program compiler(input, output, obj);

#include "globals.i"
#include "scanner.i"
```

```
#include "parser.i"
#include "semantics.i"

begin
    { main program }
end.
```

At the point the **include** pseudo-statement is encountered in the input, the lines from the included file are interpolated into the input stream. For the purposes of translation and run-time diagnostics and statement numbers in the listings and post-mortem backtraces, the lines in the included file are numbered from 1. Nested includes are possible up to 10 deep.

See the descriptions of the **i** option of *pi* in section 5.2 above; this can be used to control listing when **include** files are present.

When a non-trivial line is encountered in the source text after an **include** finishes, the 'popped' filename is printed, in the same manner as above.

For the purposes of error diagnostics when not making a listing, the filename will be printed before each diagnostic if the current filename has changed since the last filename was printed.

5.10. Separate Compilation with Pc

A separate compilation facility is provided with the Berkeley Pascal compiler, *pc*. This facility allows programs to be divided into a number of files and the pieces to be compiled individually, to be linked together at some later time. This is especially useful for large programs, where small changes would otherwise require time-consuming re-compilation of the entire program.

Normally, *pc* expects to be given entire Pascal programs. However, if given the **-c** option on the command line, it will accept a sequence of definitions and declarations, and compile them into a **.o** file, to be linked with a Pascal program at a later time. In order that procedures and functions be available across separately compiled files, they must be declared with the directive **external**. This directive is similar to the directive **forward** in that it must precede the resolution of the function or procedure, and formal parameters and function result types must be specified at the **external** declaration and may not be specified at the resolution.

Type checking is performed across separately compiled files. Since Pascal type defintions define unique types, any types which are shared between separately compiled files must be the same definition. This seemingly impossible problem is solved using a facility similar to the **include** facility discussed above. Definitions may be placed in files with the extension **.h** and the files included by separately compiled files. Each definition from a **.h** file defines a unique type, and all uses of a definition from the same **.h** file define the same type. Similarly, the facility is extended to allow the definition of **const**s and the declaration of **label**s, **var**s, and **external function**s and **procedure**s. Thus **procedure**s and **function**s which are used between separately compiled files must be declared **external**, and must be so declared in a **.h** file included by any file which calls or resolves the **function** or **procedure**. Conversely, **function**s and **procedure**s declared **external** may only be so declared in **.h** files. These files may be included only at the outermost level, and thus define or declare global objects. Note that since only **external function** and **procedure** declarations (and not resolutions) are allowed in **.h** files, statically nested **function**s and **procedure**s can not be declared **external**.

An example of the use of included **.h** files in a program would be:

```
program compiler(input, output, obj);

#include "globals.h"
#include "scanner.h"
```

```
#include "parser.h"
#include "semantics.h"

begin
    { main program }
end.
```

This might include in the main program the definitions and declarations of all the global **labels**, **consts**, **types** **vars** from the file globals.h, and the **external function** and **procedure** declarations for each of the separately compiled files for the scanner, parser and semantics. The header file *scanner.h* would contain declarations of the form:

```
type
    token = record
        { token fields }
    end;

function scan(var inputfile: text): token;
    external;
```


Then the scanner might be in a separately compiled file containing:

```
#include "globals.h"
#include "scanner.h"

function scan;
begin
    { scanner code }
end;
```

which includes the same global definitions and declarations and resolves the scanner functions and procedures declared **external** in the file scanner.h.

A. Appendix to Wirth's Pascal Report

This section is an appendix to the definition of the Pascal language in Niklaus Wirth's *Pascal Report* and, with that Report, precisely defines the Berkeley implementation. This appendix includes a summary of extensions to the language, gives the ways in which the undefined specifications were resolved, gives limitations and restrictions of the current implementation, and lists the added functions and procedures available. It concludes with a list of differences with the commonly available Pascal 6000-3.4 implementation, and some comments on standard and portable Pascal.

A.1. Extensions to the language Pascal

This section defines non-standard language constructs available in Berkeley Pascal. The **s** standard Pascal option of the translators *pi* and *pc* can be used to detect these extensions in programs which are to be transported.

String padding

Berkeley Pascal will pad constant strings with blanks in expressions and as value parameters to make them as long as is required. The following is a legal Berkeley Pascal program:

```
program x(output);
var z : packed array [ 1 .. 13 ] of char;
begin
    z := 'red';
```

```
        writeln(z)
    end;
```

The padded blanks are added on the right. Thus the assignment above is equivalent to:

```
z := 'red
```

which is standard Pascal.

Octal constants, octal and hexadecimal write

Octal constants may be given as a sequence of octal digits followed by the character 'b' or 'B'. The forms

```
write(a:n oct)
```

and

```
write(a:n hex)
```

cause the internal representation of expression *a,* which must be Boolean, character, integer, pointer, or a user-defined enumerated type, to be written in octal or hexadecimal respectively.

Assert statement

An **assert** statement causes a *Boolean* expression to be evaluated each time the statement is executed. A runtime error results if any of the expressions evaluates to be *false*. The **assert** statement is treated as a comment if run-time tests are disabled. The syntax for **assert** is:

```
assert <expr>
```

Enumerated type input-output

Enumerated types may be read and written. On output the string name associated with the enumerated value is output. If the value is out of range, a runtime error occurs. On input an identifier is read and looked up in a table of names associated with the type of the variable, and the appropriate internal value is assigned to the variable being read. If the name is not found in the table a runtime error occurs.

Structure returning functions

An extension has been added which allows functions to return arbitrary sized structures rather than just scalars as in the standard.

Separate compilation

The compiler *pc* has been extended to allow separate compilation of programs. Procedures and functions declared at the global level may be compiled separately. Type checking of calls to separately compiled routines is performed at load time to insure that the program as a whole is consistent. See section 5.10 for details.

A.2. Resolution of the undefined specifications

File name – file variable associations

Each Pascal file variable is associated with a named UNIX file. Except for *input* and *output,* which are exceptions to some of the rules, a name can become associated with a file in any of three ways:

1) If a global Pascal file variable appears in the **program** statement then it is associated with UNIX file of the same name.

2) If a file was reset or rewritten using the extended two-argument form of *reset* or *rewrite* then the given name is associated.

3) If a file which has never had UNIX name associated is reset or rewritten without specifying a name via the second argument, then a temporary name of the form 'tmp.x' is associated with the file. Temporary names start with 'tmp.1' and continue by incrementing the last character in the USASCII ordering. Temporary files are removed automatically when their scope is exited.

The program statement

The syntax of the **program** statement is:

program <id> (<file id> { , <file id > }) ;

The file identifiers (other than *input* and *output*) must be declared as variables of **file** type in the global declaration part.

The files input and output

The formal parameters *input* and *output* are associated with the UNIX standard input and output and have a somewhat special status. The following rules must be noted:

1) The program heading **must** contains the formal parameter *output.* If *input* is used, explicitly or implicitly, then it must also be declared here.

2) Unlike all other files, the Pascal files *input* and *output* must not be defined in a declaration, as their declaration is automatically:

```
var input, output: text
```

3) The procedure *reset* may be used on *input.* If no UNIX file name has ever been associated with *input,* and no file name is given, then an attempt will be made to 'rewind' *input.* If this fails, a run time error will occur. *Rewrite* calls to output act as for any other file, except that *output* initially has no associated file. This means that a simple

```
rewrite(output)
```

associates a temporary name with *output.*

Details for files

If a file other than *input* is to be read, then reading must be initiated by a call to the procedure *reset* which causes the Pascal system to attempt to open the associated UNIX file for reading. If this fails, then a runtime error occurs. Writing of a file other than *output* must be initiated by a *rewrite* call, which causes the Pascal system to create the associated UNIX file and to then open the file for writing only.

Buffering

The buffering for *output* is determined by the value of the **b** option at the end of the **program** statement. If it has its default value 1, then *output* is buffered in blocks of up to 512 characters, flushed whenever a writeln occurs and at each reference to the file *input.* If it has the value 0, *output* is unbuffered. Any value of 2 or more gives block buffering without line or *input* reference flushing. All other output files are always buffered in blocks of 512 characters. All output buffers are flushed when the files are closed at scope exit, whenever the procedure *message* is called, and can be flushed using the built-in procedure *flush.*

An important point for an interactive implementation is the definition of 'input↑'. If *input* is a teletype, and the Pascal system reads a character at the beginning of execution to define 'input↑', then no prompt could be printed by the program before the user is required to type some input. For this reason, 'input↑' is not defined by the system until its definition is needed, reading from a file occurring only when necessary.

The character set

Seven bit USASCII is the character set used on UNIX. The standard Pascal symbols 'and', 'or', 'not', '<=', '>=', '<>', and the uparrow '↑' (for pointer qualification) are recognized.† Less portable are the synonyms tilde '~' for **not**, '&' for **and**, and '|' for **or**.

Upper and lower case are considered to be distinct. Keywords and built-in **procedure** and **function** names are composed of all lower case letters. Thus the identifiers GOTO and GOto are distinct both from each other and from the keyword **goto**. The standard type 'boolean' is also available as 'Boolean'.

Character strings and constants may be delimited by the character '' or by the character '#'; the latter is sometimes convenient when programs are to be transported. Note that the '#' character has special meaning when it is the first character on a line – see *Multi-file programs* below.

The standard types

The standard type *integer* is conceptually defined as

```
type integer = minint .. maxint;
```

Integer is implemented with 32 bit twos complement arithmetic. Predefined constants of type *integer* are:

```
const maxint = 2147483647; minint = -2147483648;
```

The standard type *char* is conceptually defined as

```
type char = minchar .. maxchar;
```

Built-in character constants are 'minchar' and 'maxchar', 'bell' and 'tab'; ord(minchar) = 0, ord(maxchar) = 127.

The type *real* is implemented using 64 bit floating point arithmetic. The floating point arithmetic is done in 'rounded' mode, and provides approximately 17 digits of precision with numbers as small as 10 to the negative 38th power and as large as 10 to the 38th power.

Comments

Comments can be delimited by either '{' and '}' or by '(*' and '*)'. If the character '{' appears in a comment delimited by '{' and '}', a warning diagnostic is printed. A similar warning will be printed if the sequence '(*' appears in a comment delimited by '(*' and '*)'. The restriction implied by this warning is not part of standard Pascal, but detects many otherwise subtle errors.

†On many terminals and printers, the up arrow is represented as a circumflex '^'. These are not distinct characters, but rather different graphic representations of the same internal codes.
The proposed standard for Pascal considers them to be the same.

Option control

Options of the translators may be controlled in two distinct ways. A number of options may appear on the command line invoking the translator. These options are given as one or more strings of letters preceded by the character '–' and cause the default setting of each given option to be changed. This method of communication of options is expected to predominate for UNIX. Thus the command

```
% pi -l -s foo.p
```

translates the file foo.p with the listing option enabled (as it normally is off), and with only standard Pascal features available.

If more control over the portions of the program where options are enabled is required, then option control in comments can and should be used. The format for option control in comments is identical to that used in Pascal 6000–3.4. One places the character '$' as the first character of the comment and follows it by a comma separated list of directives. Thus an equivalent to the command line example given above would be:

```
{$l+,s+ listing on, standard Pascal}
```

as the first line of the program. The 'l' option is more appropriately specified on the command line, since it is extremely unlikely in an interactive environment that one wants a listing of the program each time it is translated.

Directives consist of a letter designating the option, followed either by a '+' to turn the option on, or by a '–' to turn the option off. The **b** option takes a single digit instead of a '+' or '–'.

Notes on the listings

The first page of a listing includes a banner line indicating the version and date of generation of *pi* or *pc*. It also includes the UNIX path name supplied for the source file and the date of last modification of that file.

Within the body of the listing, lines are numbered consecutively and correspond to the line numbers for the editor. Currently, two special kinds of lines may be used to format the listing: a line consisting of a form-feed character, control-l, which causes a page eject in the listing, and a line with no characters which causes the line number to be suppressed in the listing, creating a truly blank line. These lines thus correspond to 'eject' and 'space' macros found in many assemblers. Non-printing characters are printed as the character '?' in the listing.†

The standard procedure write

If no minimum field length parameter is specified for a *write,* the following default values are assumed:

integer	10
real	22
Boolean	length of 'true' or 'false'
char	1
string	length of the string
oct	11
hex	8

The end of each line in a text file should be explicitly indicated by 'writeln(f)', where

†The character generated by a control-i indents to the next 'tab stop'. Tab stops are set every 8 columns in UNIX. Tabs thus provide a quick way of indenting in the program.

'writeln(output)' may be written simply as 'writeln'. For UNIX, the built-in function 'page(f)' puts a single ASCII form-feed character on the output file. For programs which are to be transported the filter *pcc* can be used to interpret carriage control, as UNIX does not normally do so.

A.3. Restrictions and limitations

Files

Files cannot be members of files or members of dynamically allocated structures.

Arrays, sets and strings

The calculations involving array subscripts and set elements are done with 16 bit arithmetic. This restricts the types over which arrays and sets may be defined. The lower bound of such a range must be greater than or equal to –32768, and the upper bound less than 32768. In particular, strings may have any length from 1 to 65535 characters, and sets may contain no more than 65535 elements.

Line and symbol length

There is no intrinsic limit on the length of identifiers. Identifiers are considered to be distinct if they differ in any single position over their entire length. There is a limit, however, on the maximum input line length. This limit is quite generous however, currently exceeding 160 characters.

Procedure and function nesting and program size

At most 20 levels of **procedure** and **function** nesting are allowed. There is no fundamental, translator defined limit on the size of the program which can be translated. The ultimate limit is supplied by the hardware and thus, on the PDP-11, by the 16 bit address space. If one runs up against the 'ran out of memory' diagnostic the program may yet translate if smaller procedures are used, as a lot of space is freed by the translator at the completion of each **procedure** or **function** in the current implementation.

On the VAX-11, there is an implementation defined limit of 65536 bytes per variable. There is no limit on the number of variables.

Overflow

There is currently no checking for overflow on arithmetic operations at run-time on the PDP-11. Overflow checking is performed on the VAX-11 by the hardware.

A.4. Added types, operators, procedures and functions

Additional predefined types

The type *alfa* is predefined as:

```
type alfa = packed array [ 1..10 ] of char
```

The type *intset* is predefined as:

```
type intset = set of 0..127
```

In most cases the context of an expression involving a constant set allows the translator to determine the type of the set, even though the constant set itself may not uniquely determine this type. In the cases where it is not possible to determine the type of the set from local context, the expression type defaults to a set over the entire base type unless the base type is integer†. In the latter case the type defaults to the current binding of *intset,* which must be

†The current translator makes a special case of the construct 'if ... in [...]' and enforces only the more lax restriction on 16 bit arithmetic given above in this case.

"type set of (a subrange of) integer" at that point.

Note that if *intset* is redefined via:

type intset = **set of** 0..58;

then the default integer set is the implicit *intset* of Pascal 6000-3.4

Additional predefined operators

The relationals '<' and '>' of proper set inclusion are available. With a and b sets, note that

(**not** (a < b)) <> (a >= b)

As an example consider the sets a = [0,2] and b = [1]. The only relation true between these sets is '<>'.

Non-standard procedures

argv(i,a)	where i is an integer and a is a string variable assigns the (possibly truncated or blank padded) i 'th argument of the invocation of the current UNIX process to the variable a. The range of valid i is *0* to *argc-1*.
date(a)	assigns the current date to the alfa variable a in the format 'dd mmm yy ', where 'mmm' is the first three characters of the month, i.e. 'Apr'.
flush(f)	writes the output buffered for Pascal file f into the associated UNIX file.
halt	terminates the execution of the program with a control flow backtrace.
linelimit(f,x)‡	with f a textfile and x an integer expression causes the program to be abnormally terminated if more than x lines are written on file f. If x is less than 0 then no limit is imposed.
message(x,...)	causes the parameters, which have the format of those to the built-in **procedure** *write,* to be written unbuffered on the diagnostic unit 2, almost always the user's terminal.
null	a procedure of no arguments which does absolutely nothing. It is useful as a place holder, and is generated by *pxp* in place of the invisible empty statement.
remove(a)	where a is a string causes the UNIX file whose name is a, with trailing blanks eliminated, to be removed.
reset(f,a)	where a is a string causes the file whose name is a (with blanks trimmed) to be associated with f in addition to the normal function of *reset.*
rewrite(f,a)	is analogous to 'reset' above.
stlimit(i)	where i is an integer sets the statement limit to be i statements. Specifying the **p** option to *pc* disables statement limit counting.
time(a)	causes the current time in the form ' hh:mm:ss ' to be assigned to the alfa variable a.

‡Currently ignored by pdp-11 *px*.

Non-standard functions

argc	returns the count of arguments when the Pascal program was invoked. *Argc* is always at least 1.
card(x)	returns the cardinality of the set *x*, i.e. the number of elements contained in the set.
clock	returns an integer which is the number of central processor milliseconds of user time used by this process.
expo(x)	yields the integer valued exponent of the floating-point representation of *x*; expo(*x*) = entier(log2(abs(*x*))).
random(x)	where *x* is a real parameter, evaluated but otherwise ignored, invokes a linear congruential random number generator. Successive seeds are generated as (seed*a + c) mod m and the new random number is a normalization of the seed to the range 0.0 to 1.0; a is 62605, c is 113218009, and m is 536870912. The initial seed is 7774755.
seed(i)	where *i* is an integer sets the random number generator seed to *i* and returns the previous seed. Thus seed(seed(i)) has no effect except to yield value *i*.
sysclock	an integer function of no arguments returns the number of central processor milliseconds of system time used by this process.
undefined(x)	a Boolean function. Its argument is a real number and it always returns false.
wallclock	an integer function of no arguments returns the time in seconds since 00:00:00 GMT January 1, 1970.

A.5. Remarks on standard and portable Pascal

It is occasionally desirable to prepare Pascal programs which will be acceptable at other Pascal installations. While certain system dependencies are bound to creep in, judicious design and programming practice can usually eliminate most of the non-portable usages. Wirth's *Pascal Report* concludes with a standard for implementation and program exchange.

In particular, the following differences may cause trouble when attempting to transport programs between this implementation and Pascal 6000-3.4. Using the s translator option may serve to indicate many problem areas.†

Features not available in Berkeley Pascal

Segmented files and associated functions and procedures.

The function *trunc* with two arguments.

Arrays whose indices exceed the capacity of 16 bit arithmetic.

Features available in Berkeley Pascal but not in Pascal 6000-3.4

The procedures *reset* and *rewrite* with file names.

The functions *argc, seed, sysclock,* and *wallclock.*

The procedures *argv, flush,* and *remove.*

Message with arguments other than character strings.

---

†The s option does not, however, check that identifiers differ in the first 8 characters. *Pi* and *pc* also do not check the semantics of **packed**.

Write with keyword **hex**.

The **assert** statement.

Reading and writing of enumerated types.

Allowing functions to return structures.

Separate compilation of programs.

Comparison of records.

Other problem areas

Sets and strings are more general in Berkeley Pascal; see the restrictions given in the Jensen-Wirth *User Manual* for details on the 6000–3.4 restrictions.

The character set differences may cause problems, especially the use of the function *chr*, characters as arguments to *ord*, and comparisons of characters, since the character set ordering differs between the two machines.

The Pascal 6000–3.4 compiler uses a less strict notion of type equivalence. In Berkeley Pascal, types are considered identical only if they are represented by the same type identifier. Thus, in particular, unnamed types are unique to the variables/fields declared with them.

Pascal 6000–3.4 doesn't recognize our option flags, so it is wise to put the control of Berkeley Pascal options to the end of option lists or, better yet, restrict the option list length to one.

For Pascal 6000–3.4 the ordering of files in the program statement has significance. It is desirable to place *input* and *output* as the first two files in the **program** statement.

Acknowledgments

The financial support of William Joy and Susan Graham by the National Science Foundation under grants MCS74-07644-A04, MCS78-07291, and MCS80-05144, and the William Joy by an IBM Graduate Fellowship are gratefully acknowledged.

Berkeley VAX/UNIX Assembler Reference Manual

John F. Reiser
Bell Laboratories,
Holmdel, NJ

and

Robert R. Henry[1]
Electronics Research Laboratory
University of California
Berkeley, CA 94720

November 5, 1979

Revised
February 9, 1983

1. Introduction

This document describes the usage and input syntax of the UNIX VAX-11 assembler *as*. *As* is designed for assembling the code produced by the "C" compiler; certain concessions have been made to handle code written directly by people, but in general little sympathy has been extended. This document is intended only for the writer of a compiler or a maintainer of the assembler.

1.1. Assembler Revisions since November 5, 1979

There has been one major change to *as* since the last release. *As* has been updated to assemble the new instructions and data formats for "G" and "H" floating point numbers, as well as the new queue instructions.

PS1 5

1.2. Features Supported, but No Longer Encouraged as of February 9, 1983

These feature(s) in *as* are supported, but no longer encouraged.

- The colon operator for field initialization is likely to disappear.

2. Usage

As is invoked with these command arguments:

as [**-LVWJR**] [**-d***n*] [**-DTS**] [**-t** *directory*] [**-o** *output*] [$name_1$] ··· [$name_n$]

The **-L** flag instructs the assembler to save labels beginning with a "L" in the symbol table portion of the *output* file. Labels are not saved by default, as the default action of the link editor *ld* is to discard them anyway.

The **-V** flag tells the assembler to place its interpass temporary file into virtual memory. In normal circumstances, the system manager will decide where the temporary file should lie. Our experiments with very large temporary files show that placing the temporary file into virtual memory will save about 13% of the assembly time, where the size of the temporary file is about 350K bytes. Most assembler sources will not be this long.

[1]Preparation of this paper supported in part by the National Science Foundation under grant MCS #78-07291.

The -W turns of all warning error reporting.

The **-J** flag forces UNIX style pseudo-branch instructions with destinations further away than a byte displacement to be turned into jump instructions with 4 byte offsets. The **-J** flag buys you nothing if **-d2** is set. (See §8.4, and future work described in §11)

The **-R** flag effectively turns "**.data** *n*" directives into "**.text** *n*" directives. This obviates the need to run editor scripts on assembler source to "read-only" fix initialized data segments. Uninitialized data (via **.lcomm** and **.comm** directives) is still assembled into the data or bss segments.

The **-d** flag specifies the number of bytes which the assembler should allow for a displacement when the value of the displacement expression is undefined in the first pass. The possible values of *n* are 1, 2, or 4; the assembler uses 4 bytes if **-d** is not specified. See §8.2.

Provided the **-V** flag is not set, the **-t** flag causes the assembler to place its single temporary file in the *directory* instead of in */tmp*.

The **-o** flag causes the output to be placed on the file *output*. By default, the output of the assembler is placed in the file *a.out* in the current directory.

The input to the assembler is normally taken from the standard input. If file arguments occur, then the input is taken sequentially from the files $name_1$, $name_2$ $\cdots$ $name_n$ This is not to say that the files are assembled separately; $name_1$ is effectively concatenated to $name_2$, so multiple definitions cannot occur amongst the input sources.

The **-D** (debug), **-T** (token trace), and the **-S** (symbol table) flags enable assembler trace information, provided that the assembler has been compiled with the debugging code enabled. The information printed is long and boring, but useful when debugging the assembler.

3. Lexical conventions

Assembler tokens include identifiers (alternatively, "symbols" or "names"), constants, and operators.

3.1. Identifiers

An identifier consists of a sequence of alphanumeric characters (including period ".", underscore "_", and dollar "$"). The first character may not be numeric. Identifiers may be (practically) arbitrary long; all characters are significant.

3.2. Constants

3.2.1. Scalar constants

All scalar (non floating point) constants are (potentially) 128 bits wide. Such constants are interpreted as two's complement numbers. Note that 64 bit (quad words) and 128 bit (octal word) integers are only partially supported by the VAX hardware. In addition, 128 bit integers are only supported by the extended VAX architecture. *As* supports 64 and 128 bit integers only so they can be used as immediate constants or to fill initialized data space. *As* can not perform arithmetic on constants larger than 32 bits.

Scalar constants are initially evaluated to a full 128 bits, but are pared down by discarding high order copies of the sign bit and categorizing the number as a long, quad or octal integer. Numbers with less precision than 32 bits are treated as 32 bit quantities.

The digits are "0123456789abcdefABCDEF" with the obvious values.

An octal constant consists of a sequence of digits with a leading zero.

A decimal constant consists of a sequence of digits without a leading zero.

A hexadecimal constant consists of the characters "0x" (or "0X") followed by a sequence of digits.

A single-character constant consists of a single quote "'" followed by an ASCII character, including ASCII newline. The constant's value is the code for the given character.

3.2.2. Floating Point Constants

Floating point constants are internally represented in the VAX floating point format that is specified by the lexical form of the constant. Using the meta notation that [dec] is a decimal digit ("0123456789"), [expt] is a type specification character ("fFdDhHgG"), [expe] is a exponent delimiter and type specification character ("eEfFdDhHgG"), x^* means 0 or more occurences of x, x^+ means 1 or more occurences of x, then the general lexical form of a floating point number is:

0[expe]([+-])[dec]$^+$(.)([dec]*)([expt]([+-])(dec]$^+$))

The standard semantic interpretation is used for the signed integer, fraction and signed power of 10 exponent. If the exponent delimiter is specified, it must be either an "e" or "E", or must agree with the initial type specification character that is used. The type specification character specifies the type and representation of the constructed number, as follows:

type character	floating representation	size (bits)
f, F	F format floating	32
d, D	D format floating	64
g, G	G format floating	64
h, H	H format floating	128

Note that "G" and "H" format floating point numbers are not supported by all implementations of the VAX architecture. *As* does not require the augmented architecture in order to run.

The assembler uses the library routine *atof()* to convert "F" and "D" numbers, and uses its own conversion routine (derived from *atof*, and believed to be numerically accurate) to convert "G" and "H" floating point numbers.

Collectively, all floating point numbers, together with quad and octal scalars are called *Bignums*. When *as* requires a Bignum, a 32 bit scalar quantity may also be used.

3.2.3. String Constants

A string constant is defined using the same syntax and semantics as the "C" language uses. Strings begin and end with a "\"" (double quote). The DEC MACRO-32 assembler conventions for flexible string quoting is not implemented. All "C" backslash conventions are observed; the backslash conventions peculiar to the PDP-11 assembler are not observed. Strings are known by their value and their length; the assembler does not implicitly end strings with a null byte.

3.3. Operators

There are several single-character operators; see §6.1.

3.4. Blanks

Blank and tab characters may be interspersed freely between tokens, but may not be used within tokens (except character constants). A blank or tab is required to separate adjacent identifiers or constants not otherwise separated.

3.5. Scratch Mark Comments

The character "#" introduces a comment, which extends through the end of the line on which it appears. Comments starting in column 1, having the format "# *expression string*", are interpreted as an indication that the assembler is now assembling file *string* at line *expression*. Thus, one can use the "C" preprocessor on an assembly language source file, and use the *#include* and *#define* preprocessor directives. (Note that there may not be an assembler comment starting in column 1 if the assembler source is given to the "C" preprocessor, as it will be interpreted by the preprocessor in a way not intended.) Comments are otherwise ignored by the assembler.

3.6. "C" Style Comments

The assembler will recognize "C" style comments, introduced with the prologue /* and ending with the epilogue */. "C" style comments may extend across multiple lines, and are the preferred comment style to use if one chooses to use the "C" preprocessor.

4. Segments and Location Counters

Assembled code and data fall into three segments: the text segment, the data segment, and the bss segment. The UNIX operating system makes some assumptions about the content of these segments; the assembler does not. Within the text and data segments there are a number of sub-segments, distinguished by number ("**text** 0", "**text** 1", ··· "**data** 0", "**data** 1", ···). Currently there are four subsegments each in text and data. The subsegments are for programming convenience only.

Before writing the output file, the assembler zero-pads each text subsegment to a multiple of four bytes and then concatenates the subsegments in order to form the text segment; an analogous operation is done for the data segment. Requesting that the loader define symbols and storage regions is the only action allowed by the assembler with respect to the bss segment. Assembly begins in "**text** 0".

Associated with each (sub)segment is an implicit location counter which begins at zero and is incremented by 1 for each byte assembled into the (sub)segment. There is no way to explicitly reference a location counter. Note that the location counters of subsegments other than "**text** 0" and "**data** 0" behave peculiarly due to the concatenation used to form the text and data segments.

5. Statements

A source program is composed of a sequence of *statements*. Statements are separated either by new-lines or by semicolons. There are two kinds of statements: null statements and keyword statements. Either kind of statement may be preceded by one or more labels.

5.1. Named Global Labels

A global label consists of a name followed by a colon. The effect of a name label is to assign the current value and type of the location counter to the name. An error is indicated in pass 1 if the name is already defined; an error is indicated in pass 2 if the value assigned changes the definition of the label.

A global label is referenced by its name.

Global labels beginning with a "L" are discarded unless the -L option is in effect.

5.2. Numeric Local Labels

A numeric label consists of a digit 0 to 9 followed by a colon. Such a label serves to define temporary symbols of the form "*n*b" and "*n*f", where *n* is the digit of the label. As

in the case of name labels, a numeric label assigns the current value and type of the location counter to the temporary symbol. However, several numeric labels with the same digit may be used within the same assembly. References to symbols of the form "*n*b" refer to the first numeric label "*n*:" *b*ackwards from the reference; "*n*f" symbols refer to the first numeric label "*n*:" *f*orwards from the reference. Such numeric labels conserve the inventive powers of the human programmer.

For various reasons, *as* turns local labels into labels of the form L*n*.\$*m*. Although unlikely, these generated labels may conflict with programmer defined labels.

5.3. Null statements

A null statement is an empty statement ignored by the assembler. A null statement may be labeled, however.

5.4. Keyword statements

A keyword statement begins with one of the many predefined keywords known to *as*; the syntax of the remainder of the statement depends on the keyword. All instruction opcodes are keywords. The remaining keywords are assembler pseudo-operations, also called *directives*. The pseudo-operations are listed in §8, together with the syntax they require.

6. Expressions

An expression is a sequence of symbols representing a value. Its constituents are identifiers, constants, operators, and parentheses. Each expression has a type.

All operators in expressions are fundamentally binary in nature. Arithmetic is two's complement and has 32 bits of precision. *As* can not do arithmetic on floating point numbers, quad or octal precision scalar numbers. There are four levels of precedence, listed here from lowest precedence level to highest:

precedence	operators
binary	+, -
binary	|, &, ^, !
binary	*, /, %,
unary	-, ~

All operators of the same precedence are evaluated strictly left to right, except for the evaluation order enforced by parenthesis.

6.1. Expression Operators

The operators are:

operator	meaning
+	addition
-	(binary) subtraction
*	multiplication
/	division
%	modulo
-	(unary) 2's complement
&	bitwise and
|	bitwise or
^	bitwise exclusive or
!	bitwise or not
~	bitwise 1's complement
>	logical right shift
>>	logical right shift
<	logical left shift
<<	logical left shift

Expressions may be grouped by use of parentheses, "(" and ")".

6.2. Data Types

The assembler manipulates several different types of expressions. The types likely to be met explicitly are:

undefined Upon first encounter, each symbol is undefined. It may become undefined if it is assigned an undefined expression. It is an error to attempt to assemble an undefined expression in pass 2; in pass 1, it is not (except that certain keywords require operands which are not undefined).

undefined external
A symbol which is declared **.globl** but not defined in the current assembly is an undefined external. If such a symbol is declared, the link editor *ld* must be used to load the assembler's output with another routine that defines the undefined reference.

absolute An absolute symbol is defined ultimately from a constant. Its value is unaffected by any possible future applications of the link-editor to the output file.

text The value of a text symbol is measured with respect to the beginning of the text segment of the program. If the assembler output is link-edited, its text symbols may change in value since the program need not be the first in the link editor's output. Most text symbols are defined by appearing as labels. At the start of an assembly, the value of "." is "text 0".

data The value of a data symbol is measured with respect to the origin of the data segment of a program. Like text symbols, the value of a data symbol may change during a subsequent link-editor run since previously loaded programs may have data segments. After the first **.data** statement, the value of "." is "data 0".

bss The value of a bss symbol is measured from the beginning of the bss segment of a program. Like text and data symbols, the value of a bss symbol may change during a subsequent link-editor run, since previously loaded programs may have bss segments.

external absolute, text, data, or bss
Symbols declared **.globl** but defined within an assembly as absolute, text, data,

or bss symbols may be used exactly as if they were not declared **.globl**; however, their value and type are available to the link editor so that the program may be loaded with others that reference these symbols.

register The symbols

r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 r14 r15 ap fp sp pc

are predefined as register symbols. In addition, the "%" operator converts the following absolute expression whose value is between 0 and 15 into a register reference.

other types

Each keyword known to the assembler has a type which is used to select the routine which processes the associated keyword statement. The behavior of such symbols when not used as keywords is the same as if they were absolute.

6.3. Type Propagation in Expressions

When operands are combined by expression operators, the result has a type which depends on the types of the operands and on the operator. The rules involved are complex to state but were intended to be sensible and predictable. For purposes of expression evaluation the important types are

undefined
absolute
text
data
bss
undefined external
other

The combination rules are then

(1) If one of the operands is undefined, the result is undefined.

(2) If both operands are absolute, the result is absolute.

(3) If an absolute is combined with one of the "other types" mentioned above, the result has the other type. An "other type" combined with an explicitly discussed type other than absolute it acts like an absolute.

Further rules applying to particular operators are:

\+ If one operand is text-, data-, or bss-segment relocatable, or is an undefined external, the result has the postulated type and the other operand must be absolute.

\- If the first operand is a relocatable text-, data-, or bss-segment symbol, the second operand may be absolute (in which case the result has the type of the first operand); or the second operand may have the same type as the first (in which case the result is absolute). If the first operand is external undefined, the second must be absolute. All other combinations are illegal.

others

It is illegal to apply these operators to any but absolute symbols.

7. Pseudo-operations (Directives)

The keywords listed below introduce directives or instructions, and influence the later behavior of the assembler for this statement. The metanotation

[stuff]

means that 0 or more instances of the given "stuff" may appear.

Boldface tokens must appear literally; words in *italic* words are substitutable.

The pseudo-operations listed below are grouped into functional categories.

7.1. Interface to a Previous Pass

.ABORT

As soon as the assembler sees this directive, it ignores all further input (but it does read to the end of file), and aborts the assembly. No files are created. It is anticipated that this would be used in a pipe interconnected version of a compiler, where the first major syntax error would cause the compiler to issue this directive, saving unnecessary work in assembling code that would have to be discarded anyway.

.file *string*

This directive causes the assembler to think it is in file *string*, so error messages reflect the proper source file.

.line *expression*

This directive causes the assembler to think it is on line *expression* so error messages reflect the proper source file.

The only effect of assembling multiple files specified in the command string is to insert the *file* and *line* directives, with the appropriate values, at the beginning of the source from each file.

\# *expression string*
\# *expression*

This is the only instance where a comment is meaningful to the assembler. The "#" *must* be in the first column. This meta comment causes the assembler to believe it is on line *expression*. The second argument, if included, causes the assembler to believe it is in file *string*, otherwise the current file name does not change.

7.2. Location Counter Control

.data [*expression*]
.text [*expression*]

These two pseudo-operations cause the assembler to begin assembling into the indicated text or data subsegment. If specified, the *expression* must be defined and absolute; an omitted *expression* is treated as zero. The effect of a **.data** directive is treated as a **.text** directive if the **-R** assembly flag is set. Assembly starts in the **.text** 0 subsegment.

The directives **.align** and **.org** also control the placement of the location counter.

7.3. Filled Data

.align *align_expr* [, *fill_expr*]

The location counter is adjusted so that the *expression* lowest bits of the location counter become zero. This is done by assembling from 0 to 2^{align_expr} bytes, taken from the

low order byte of *fill_expr*. If present, *fill_expr* must be absolute; otherwise it defaults to 0. Thus ".align 2" pads by null bytes to make the location counter evenly divisible by 4. The *align_expr* must be defined, absolute, nonnegative, and less than 16.

Warning: the subsegment concatenation convention and the current loader conventions may not preserve attempts at aligning to more than 2 low-order zero bits.

.org *org_expr* [, *fill_expr*]

The location counter is set equal to the value of *org_expr*, which must be defined and absolute. The value of the *org_expr* must be greater than the current value of the location counter. Space between the current value of the location counter and the desired value are filled with bytes taken from the low order byte of *fill_expr*, which must be absolute and defaults to 0.

.space *space_expr* [, *fill_expr*]

The location counter is advanced by *space_expr* bytes. *Space_expr* must be defined and absolute. The space is filled in with bytes taken from the low order byte of *fill_expr*, which must be defined and absolute. *Fill_expr* defaults to 0. The **.fill** directive is a more general way to accomplish the **.space** directive.

.fill *rep_expr*, *size_expr*, *fill_expr*

All three expressions must be absolute. *fill_expr*, treated as an expression of size *size_expr* bytes, is assembled and replicated *rep_expr* times. The effect is to advance the current location counter *rep_expr* * *size_expr* bytes. *size_expr* must be between 1 and 8.

7.4. Symbol Definitions

7.5. Initialized Data

.byte *expr* [, *expr*]
.word *expr* [, *expr*]
.int *expr* [, *expr*]
.long *expr* [, *expr*]

The *expressions* in the comma-separated list are truncated to the size indicated by the key word:

keyword	length (bits)
.byte	8
.word	16
.int	32
.long	32

and assembled in successive locations. The *expressions* must be absolute.

Each *expression* may optionally be of the form:

$$expression_1 : expression_2$$

In this case, the value of $expression_2$ is truncated to $expression_1$ bits, and assembled in the next $expression_1$ bit field which fits in the natural data size being assembled. Bits which are skipped because a field does not fit are filled with zeros. Thus, "**.byte** 123" is equivalent to

"**.byte** 8:123", and "**.byte** 3:1,2:1,5:1" assembles two bytes, containing the values 9 and 1.

NB: Bit field initialization with the colon operator is likely to disappear in future releases of the assembler.

.quad *number* [, *number*]
.octa *number* [, *number*]
.float *number* [, *number*]
.double *number* [, *number*]
.ffloat *number* [, *number*]
.dfloat *number* [, *number*]
.gfloat *number* [, *number*]
.hfloat *number* [, *number*]

These initialize Bignums (see §3.2.2) in successive locations whose size is a function on the key word. The type of the Bignums (determined by the exponent field, or lack thereof) may not agree with type implied by the key word. The following table shows the key words, their size, and the data types for the Bignums they expect.

keyword	format	length (bits)	valid *number*(s)
.quad	quad scalar	64	scalar
.octa	octal scalar	128	scalar
.float	F float	32	F, D and scalar
.ffloat	F float	32	F, D and scalar
.double	D float	64	F, D and scalar
.dfloat	D float	64	F, D and scalar
.gfloat	G float	64	G scalar
.hfloat	H float	128	H scalar

As will correctly perform other floating point conversions while initializing, but issues a warning message. *As* performs all floating point initializations and conversions using only the facilities defined in the original (native) architecture.

.ascii *string* [, *string*]
.asciz *string* [, *string*]

Each *string* in the list is assembled into successive locations, with the first letter in the string being placed into the first location, etc. The **.ascii** directive will not null pad the string; the **.asciz** directive will null pad the string. (Recall that strings are known by their length, and need not be terminated with a null, and that the "C" conventions for escaping are understood.) The **.ascii** directive is identical to:
.byte $string_0$, $string_1$, ···

.comm *name*, *expression*

Provided the *name* is not defined elsewhere, its type is made "undefined external", and its value is *expression*. In fact the *name* behaves in the current assembly just like an undefined external. However, the link editor *ld* has been special-cased so that all external symbols which are not otherwise defined, and which have a non-zero value, are defined to lie in the bss segment, and enough space is left after the symbol to hold *expression* bytes.

.lcomm *name*, *expression*

expression bytes will be allocated in the bss segment and *name* assigned the location of the first byte, but the *name* is not declared as global and hence will be unknown to the link editor.

.globl *name*

This statement makes the *name* external. If it is otherwise defined (by **.set** or by appearance as a label) it acts within the assembly exactly as if the **.globl** statement were not given; however, the link editor may be used to combine this object module with other modules referring to this symbol.

Conversely, if the given symbol is not defined within the current assembly, the link editor can combine the output of this assembly with that of others which define the symbol. The assembler makes all otherwise undefined symbols external.

.set *name*, *expression*

The (*name*, *expression*) pair is entered into the symbol table. Multiple **.set** statements with the same name are legal; the most recent value replaces all previous values.

.lsym *name*, *expression*

A unique and otherwise unreferencable instance of the (*name*, *expression*) pair is created in the symbol table. The Fortran 77 compiler uses this mechanism to pass local symbol definitions to the link editor and debugger.

.stabs *string*, $expr_1$, $expr_2$, $expr_3$, $expr_4$
.stabn $expr_1$, $expr_2$, $expr_3$, $expr_4$
.stabd $expr_1$, $expr_2$, $expr_3$

The *stab* directives place symbols in the symbol table for the symbolic debugger, *sdb*[2]. A "stab" is a *s*ymbol *tab*le entry. The **.stabs** is a string stab, the **.stabn** is a stab not having a string, and the **.stabd** is a "dot" stab that implicitly references "dot", the current location counter.

The *string* in the **.stabs** directive is the name of a symbol. If the symbol name is zero, the **.stabn** directive may be used instead.

The other expressions are stored in the name list structure of the symbol table and preserved by the loader for reference by *sdb*; the value of the expressions are peculiar to formats required by *sdb*.

$expr_1$ is used as a symbol table tag (nlist field *n_type*).

$expr_2$ seems to always be zero (nlist field *n_other*).

$expr_3$ is used for either the source line number, or for a nesting level (nlist field *n_desc*).

$expr_4$ is used as tag specific information (nlist field *n_value*). In the case of the **.stabd** directive, this expression is nonexistent, and is taken to be the value of the location counter at the following instruction. Since there is no associated name for a **.stabd** directive, it can only be used in circumstances where the name is zero. The effect of a **.stabd** directive can be achieved by one of the other **.stab**x directives in the following manner:

[2]Katseff, H.P. *Sdb: A Symbol Debugger.* Bell Laboratories, Holmdel, NJ. April 12, 1979.
Katseff, H.P. *Symbol Table Format for Sdb,* File 39394, Bell Laboratories, Holmdel, NJ. March 14, 1979.

.stabn *expr*$_1$, *expr*$_2$, *expr*$_3$, LL*n*
LL*n*:

The **.stabd** directive is preferred, because it does not clog the symbol table with labels used only for the stab symbol entries.

8. Machine instructions

The syntax of machine instruction statements accepted by *as* is generally similar to the syntax of DEC MACRO-32. There are differences, however.

8.1. Character set

As uses the character "$" instead of "#" for immediate constants, and the character "*" instead of "@" for indirection. Opcodes and register names are spelled with lower-case rather than upper-case letters.

8.2. Specifying Displacement Lengths

Under certain circumstances, the following constructs are (optionally) recognized by *as* to indicate the number of bytes to allocate for the displacement used when constructing displacement and displacement deferred addressing modes:

primary	alternate	length
B`	**B^**	byte (1 byte)
W`	**W^**	word (2 bytes)
L`	**L^**	long word (4 bytes)

One can also use lower case **b**, **w** or **l** instead of the upper case letters. There must be no space between the size specifier letter and the "`" or "^". The constructs **S^** and **G^** are not recognized by *as*, as they are by the DEC MACRO-32 assembler. It is preferred to use the "^"displacement so that the "`" is not misinterpreted as the **xor** operator.

Literal values (including floating-point literals used where the hardware expects a floating-point operand) are assembled as short literals if possible, hence not needing the **S^** DEC MACRO-32 directive.

If the displacement length modifier is present, then the displacement is **always** assembled with that displacement, even if it will fit into a smaller field, or if significance is lost. If the length modifier is not present, and if the value of the displacement is known exactly in *as*'s first pass, then *as* determines the length automatically, assembling it in the shortest possible way, Otherwise, *as* will use the value specified by the **–d** argument, which defaults to 4 bytes.

8.3. case*x* Instructions

As considers the instructions **caseb**, **casel**, **casew** to have three operands. The displacements must be explicitly computed by *as*, using one or more **.word** statements.

8.4. Extended branch instructions

These opcodes (formed in general by substituting a "j" for the initial "b" of the standard opcodes) take as branch destinations the name of a label in the current subsegment. It is an error if the destination is known to be in a different subsegment, and it is a warning if the destination is not defined within the object module being assembled.

If the branch destination is close enough, then the corresponding short branch "b" instruction is assembled. Otherwise the assembler choses a sequence of one or more

instructions which together have the same effect as if the "b" instruction had a larger span. In general, *as* chooses the inverse branch followed by a **brw**, but a **brw** is sometimes pooled among several "j" instructions with the same destination.

As is unable to perform the same long/short branch generation for other instructions with a fixed byte displacement, such as the **sob**, **aob** families, or for the **acbx** family of instructions which has a fixed word displacement. This would be desirable, but is prohibitive because of the complexity of these instructions.

If the **-J** assembler option is given, a **jmp** instruction is used instead of a **brw** instruction for **ALL** "j" instructions with distant destinations. This makes assembly of large (>32K bytes) programs (inefficiently) possible. *As* does not try to use clever combinations of **brb**, **brw** and **jmp** instructions. The **jmp** instructions use PC relative addressing, with the length of the offset given by the **-d** assembler option.

These are the extended branch instructions *as* recognizes:

jeql	**jeqlu**	**jneq**
jgeq	**jgequ**	**jgtr**
jleq	**jlequ**	**jlss**
jbcc	**jbsc**	**jbcs**
jlbc	**jlbs**	
jcc	**jcs**	
jvc	**jvs**	
jbc	**jbs**	
jbr		

Note that **jbr** turns into **brb** if its target is close enough; otherwise a **brw** is used.

9. Diagnostics

Diagnostics are intended to be self explanatory and appear on the standard output. Diagnostics either report an *error* or a *warning*. Error diagnostics complain about lexical, syntactic and some semantic errors, and abort the assembly.

The majority of the warnings complain about the use of VAX features not supported by all implementations of the architecture. *As* will warn if new opcodes are used, if "G" or "H" floating point numbers are used and will complain about mixed floating conversions.

10. Limits

limit	what
Arbitrary[3]	Files to assemble
BUFSIZ	Significant characters per name
Arbitrary	Characters per input line
Arbitrary	Characters per string
Arbitrary	Symbols
4	Text segments
4	Data segments

[3]Although the number of characters available to the *argv* line is restricted by UNIX to 10240.

11. Annoyances and Future Work

Most of the annoyances deal with restrictions on the extended branch instructions.

As only uses a two level algorithm for resolving extended branch instructions into short or long displacements. What is really needed is a general mechanism to turn a short conditional jump into a reverse conditional jump over one of **two** possible unconditional branches, either a **brw** or a **jmp** instruction. Currently, the **-J** forces the **jmp** instruction to *always* be used, instead of the shorter **brw** instruction when needed.

The assembler should also recognize extended branch instructions for **sob**, **aob**, and **acbx** instructions. **Sob** instructions will be easy, **aob** will be harder because the synthesized instruction uses the index operand twice, so one must be careful of side effects, and the **acbx** family will be much harder (in the general case) because the comparison depends on the sign of the addend operand, and two operands are used more than once. Augmenting *as* with these extended loop instructions will allow the peephole optimizer to produce much better loop optimizations, since it currently assumes the worst case about the size of the loop body.

The string temporary file is not put in memory when the -V flag is set. The string table in the generated a.out contains some strings and names that are never referenced from the symbol table; the loader removes these unreferenced strings, however.

Berkeley Software Architecture Manual
4.3BSD Edition

William Joy, Robert Fabry,

Samuel Leffler, M. Kirk McKusick,

Michael Karels

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

ABSTRACT

This document summarizes the facilities provided by the 4.3BSD version of the UNIX* operating system. It does not attempt to act as a tutorial for use of the system nor does it attempt to explain or justify the design of the system facilities. It gives neither motivation nor implementation details, in favor of brevity.

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

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

* UNIX is a trademark of Bell Laboratories.

TABLE OF CONTENTS

2. System facilities

2.1. Generic operations
2.1.1. Read and write
2.1.2. Input/output control
2.1.3. Non-blocking and asynchronous operations

2.2. File system
2.2.1 Overview
2.2.2. Naming
2.2.3. Creation and removal
2.2.3.1. Directory creation and removal
2.2.3.2. File creation
2.2.3.3. Creating references to devices
2.2.3.4. Portal creation
2.2.3.6. File, device, and portal removal
2.2.4. Reading and modifying file attributes
2.2.5. Links and renaming
2.2.6. Extension and truncation
2.2.7. Checking accessibility
2.2.8. Locking
2.2.9. Disc quotas

2.3. Interprocess communication
2.3.1. Interprocess communication primitives
2.3.1.1. Communication domains
2.3.1.2. Socket types and protocols
2.3.1.3. Socket creation, naming and service establishment
2.3.1.4. Accepting connections
2.3.1.5. Making connections
2.3.1.6. Sending and receiving data
2.3.1.7. Scatter/gather and exchanging access rights
2.3.1.8. Using read and write with sockets
2.3.1.9. Shutting down halves of full-duplex connections
2.3.1.10. Socket and protocol options
2.3.2. UNIX domain
2.3.2.1. Types of sockets
2.3.2.2. Naming
2.3.2.3. Access rights transmission
2.3.3. INTERNET domain
2.3.3.1. Socket types and protocols
2.3.3.2. Socket naming
2.3.3.3. Access rights transmission
2.3.3.4. Raw access

2.4. Terminals and devices
2.4.1. Terminals
2.4.1.1. Terminal input
2.4.1.1.1 Input modes
2.4.1.1.2 Interrupt characters
2.4.1.1.3 Line editing
2.4.1.2. Terminal output
2.4.1.3. Terminal control operations
2.4.1.4. Terminal hardware support
2.4.2. Structured devices

2.4.3. Unstructured devices

2.5. Process control and debugging

I. Summary of facilities

0. Notation and types

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

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

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

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

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

1. Kernel primitives

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

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

1.1. Processes and protection

1.1.1. Host and process identifiers

Each UNIX host has associated with it a 32-bit host id, and a host name of up to 64 characters (as defined by MAXHOSTNAMELEN in *<sys/param.h>*). These are set (by a privileged user) and returned by the calls:

```
sethostid(hostid)
long hostid;

hostid = gethostid();
result long hostid;

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

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

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

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

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

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

1.1.2. Process creation and termination

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

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

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

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

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

returning 8 bits of exit status to its parent.

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

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

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

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

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

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

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

1.1.3. User and group ids

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

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

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

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

the real and effective accounting group ids by:

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

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

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

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

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

* The type of the gidset array in getgroups and setgroups remains integer for compatibility with 4.2BSD. It may change to **gid_t** in future releases.

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

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

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

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

1.1.4. Process groups

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

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

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

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

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

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

1.2. Memory management†

1.2.1. Text, data and stack

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

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

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

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

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

1.2.2. Mapping pages

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

```
/* protections are chosen from these bits, or-ed together */
#define  PROT_READ          0x04    /* pages can be read */
#define  PROT_WRITE         0x02    /* pages can be written */
#define  PROT_EXEC          0x01    /* pages can be executed */

/* flags contain mapping type, sharing type and options */
/* mapping type; choose one */
#define MAP_FILE            0x0001  /* mapped from a file or device */
#define MAP_ANON            0x0002  /* allocated from memory, swap space */
#define MAP_TYPE            0x000f  /* mask for type field */

/* sharing types; choose one */
#define  MAP_SHARED         0x0010  /* share changes */
#define  MAP_PRIVATE        0x0000  /* changes are private */

/* other flags */
#define MAP_FIXED           0x0020  /* map addr must be exactly as requested */
#define MAP_NOEXTEND        0x0040  /* for MAP_FILE, don't change file size */
#define MAP_HASSEMPHORE     0x0080  /* region may contain semaphores */
#define MAP_INHERIT         0x0100  /* region is retained after exec */
```

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

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

The call:

† This section represents the interface planned for later releases of the system. Of the calls described in this section, only *sbrk* and *getpagesize* are included in 4.3BSD.

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

causes the pages starting at *addr* and continuing for at most *len* bytes to be mapped from the object represented by descriptor *fd*, starting at byte offset *pos*. The starting address of the region is returned; for the convenience of the system, it may be different than that supplied unless the MAP_FIXED flag is given, in which case the exact address will be used or the call will fail. The actual amount mapped is returned in *len*. The *addr*, *len*, and *pos* parameters must all be multiples of the pagesize. The parameter *prot* specifies the accessibility of the mapped pages. The parameter *flags* specifies the type of object to be mapped, mapping options, and whether modifications made to this mapped copy of the page are to be kept *private*, or are to be *shared* with other references. Possible types include MAP_FILE, mapping a regular file or character-special device memory, and MAP_ANON, which maps memory not associated with any specific file. The file descriptor used for creating MAP_ANON regions is used only for naming, and may be given as −1 if no name is associated with the region‡. The MAP_NOEXTEND flag prevents the mapped file from being extended despite rounding due to the granularity of mapping. The MAP_HASSEMAPHORE flag allows special handling for regions that may contain semaphores. The MAP_INHERIT flag allows a region to be inherited after an *exec*.

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

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

writes any modified pages back to the filesystem and updates the file modification time. If *len* is 0, all modified pages within the region containing *addr* will be flushed; if *len* is non-zero, only the pages containing *addr* and *len* succeeding locations will be examined. Any required invalidation of memory caches will also take place at this time. Filesystem operations on a file which is mapped for shared modifications are unpredictable except after an *msync*.

A mapping can be removed by the call

```
munmap(addr);
caddr_t addr;
```

This call deletes the region containing the address given, and causes further references to addresses within the region to generate invalid memory references.

1.2.3. Page protection control

A process can control the protection of pages using the call

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

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

1.2.4. Giving and getting advice

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

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

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

‡ The current design does not allow a process to specify the location of swap space. In the future we may define an additional mapping type, MAP_SWAP, in which the file descriptor argument specifies a file or device to which swapping should be done.

```
#define MADV_NORMAL     0   /* no further special treatment */
#define MADV_RANDOM     1   /* expect random page references */
#define MADV_SEQUENTIAL 2   /* expect sequential references */
#define MADV_WILLNEED   3   /* will need these pages */
#define MADV_DONTNEED   4   /* don't need these pages */
#define MADV_SPACEAVAIL 5   /* insure that resources are reserved */
```

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

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

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

1.2.5. Synchronization primitives

Primitives are provided for synchronization using semaphores in shared memory. Semaphores must lie within a MAP_SHARED region with at least modes PROT_READ and PROT_WRITE. The MAP_HASSEMAPHORE flag must have been specified when the region was created. To acquire a lock a process calls:

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

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

To release a lock a process calls:

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

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

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

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

The process will remain in a sleeping state until some other process issues an *mwakeup* for the same semaphore within the region using the call:

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

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

1.3. Signals

1.3.1. Overview

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

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

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

1.3.2. Signal types

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

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

Software signals reflect interrupts generated by user request: SIGINT for the normal interrupt signal; SIGQUIT for the more powerful *quit* signal, that normally causes a core image to be generated; SIGHUP and SIGTERM that cause graceful process termination, either because a user has "hung up", or by user or program request; and SIGKILL, a more powerful termination signal which a process cannot catch or ignore. Programs may define their own asynchronous events using SIGUSR1 and SIGUSR2. Other software signals (SIGALRM, SIGVTALRM, SIGPROF) indicate the expiration of interval timers.

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

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

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

1.3.3. Signal handlers

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

```
#include <signal.h>

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

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

assigns interrupt handler address *sv_handler* to signal *signo*. Each handler address specifies either an interrupt routine for the signal, that the signal is to be ignored, or that a default action (usually process termination) is to occur if the signal occurs. The constants SIG_IGN and SIG_DEF used as values for *sv_handler* cause ignoring or defaulting of a condition. The *sv_mask* value specifies the signal mask to be used when the handler is invoked; it implicitly includes the signal which invoked the handler. Signal masks include one bit for each signal; the mask for a signal *signo* is provided by the macro *sigmask(signo)*, from *<signal.h>*. *Sv_flags* specifies whether system calls should be restarted if the signal handler returns and whether the handler should operate on the normal run-time stack or a special signal stack (see below). If *osv* is non-zero, the previous signal vector is returned.

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

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

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

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

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

PS1 6

1.3.4. Sending signals

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

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

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

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

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

1.3.5. Protecting critical sections

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

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

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

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

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

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

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

1.3.6. Signal stacks

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

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

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

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

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

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

1.4. Timers

1.4.1. Real time

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

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

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

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

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

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

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

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

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

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

1.4.2. Interval time

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

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

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

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

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

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

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

and a timer is set or read by the call:

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

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

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

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

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

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

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

1.5. Descriptors

1.5.1. The reference table

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

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

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

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

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

1.5.2. Descriptor properties

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

1.5.3. Managing descriptor references

A duplicate of a descriptor reference may be made by doing

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

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

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

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

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

1.5.4. Multiplexing requests

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

Synchronous multiplexing is performed by using the *select* call to examine the state of multiple descriptors simultaneously, and to wait for state changes on those descriptors. Sets of descriptors of interest are specified as bit masks, as follows:

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

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

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

The *select* call examines the descriptors specified by the sets *in*, *out* and *except*, replacing the specified bit masks by the subsets that select true for input, output, and exceptional conditions respectively (*nd* indicates the number of file descriptors specified by the bit masks). If any descriptors meet the following criteria, then the number of such descriptors is returned in *nds* and the bit masks are updated.

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

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

Options affecting I/O on a descriptor may be read and set by the call:

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

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

The F_SETFL *cmd* may be used to set a descriptor in non-blocking I/O mode and/or enable signaling when I/O is possible. F_SETOWN may be used to specify a process or process group to be signaled when using the latter mode of operation or when urgent indications arise.

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

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

1.5.5. Descriptor wrapping.†

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

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

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

Protocols may be based on communications multiplexing or a rights-passing style of handling multiple requests made on the same object. For instance, a protocol for implementing a file abstraction may or may not include locally generated "read-ahead" requests. A protocol that provides for read-ahead may provide higher performance but have a more difficult implementation.

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

† The facilities described in this section are not included in 4.3BSD.

1.6. Resource controls

1.6.1. Process priorities

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

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

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

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

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

1.6.2. Resource utilization

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

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

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

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

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

1.6.3. Resource limits

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

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

#define RLIM_NLIMITS    6

#define RLIM_INFINITY   0x7fffffff

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

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

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

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

1.7. System operation support

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

1.7.1. Bootstrap operations

The call

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

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

The call

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

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

1.7.2. Shutdown operations

The call

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

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

The call

```
sync();
```

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

The call

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

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

1.7.3. Accounting

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

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

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

2. System facilities

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

The system abstractions described are:

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

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

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

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

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

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

† Support for mapping files is not included in the 4.3 release.

2.1. Generic operations

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

2.1.1. Read and write

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

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

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

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

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

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

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

The calls using an array of descriptors are:

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

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

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

2.1.2. Input/output control

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

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

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

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

2.1.3. Non-blocking and asynchronous operations

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

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

Operations other than data input or output may be performed on a descriptor in a non-blocking fashion. These operations will return with a characteristic error indicating that they are in progress if they cannot complete immediately. The descriptor may then be *selected* for *write* to find out when the operation has been completed. When *select* indicates the descriptor is writeable, the operation has completed. Depending on the nature of the descriptor and the operation, additional activity may be started or the new state may be tested.

2.2. File system

2.2.1. Overview

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

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

2.2.2. Naming

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

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

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

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

The calls

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

chroot(path)
char *path;
```

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

2.2.3. Creation and removal

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

2.2.3.1. Directory creation and removal

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

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

where the mode is defined as for files (see below). Directories are removed with the *rmdir* system call:

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

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

2.2.3.2. File creation

Files are created with the *open* system call,

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

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

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

One of O_RDONLY, O_WRONLY and O_RDWR should be specified, indicating what types of operations are desired to be performed on the open file. The operations will be checked against the user's access rights to the file before allowing the *open* to succeed. Specifying O_APPEND causes writes to automatically append to the file. The flag O_CREAT causes the file to be created if it does not exist, owned by the current user and the group of the containing directory. The protection for the new file is specified in *mode*. The file mode is used as a three digit octal number. Each digit encodes read access as 4, write access as 2 and execute access as 1, or'ed together. The 0700 bits describe owner access, the 070 bits describe the access rights for processes in the same group as the file, and the 07 bits describe the access rights for other processes.

If the open specifies to create the file with O_EXCL and the file already exists, then the *open* will fail without affecting the file in any way. This provides a simple exclusive access facility. If the file exists but is a symbolic link, the open will fail regardless of the existence of the file specified by the link.

2.2.3.3. Creating references to devices

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

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

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

2.2.3.4. Portal creation†

The call

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

† The *portal* call is not implemented in 4.3BSD.

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

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

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

2.2.3.5. File, device, and portal removal

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

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

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

2.2.4. Reading and modifying file attributes

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

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

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

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

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

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

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

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

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

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

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

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

where *mode* is a value indicating the new protection of the file, as listed in section 2.2.3.2.

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

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

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

2.2.5. Links and renaming

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2.6. Extension and truncation

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

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

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

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

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

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

A file may be truncated with either of the calls:

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

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

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

2.2.7. Checking accessibility

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

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

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

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

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

2.2.8. Locking

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

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

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

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

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

Successive lock calls may be used to increase or decrease the level of locking. If an object is currently locked by another process when a *flock* call is made, the caller will be blocked until the current lock owner releases the lock; this may be avoided by including LOCK_NB in the *how* parameter. Specifying LOCK_UN removes all locks associated with the descriptor. Advisory locks held by a process are automatically deleted when the process terminates.

2.2.9. Disk quotas

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

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

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

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

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

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

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

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

2.3. Interprocess communications

2.3.1. Interprocess communication primitives

2.3.1.1. Communication domains

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

2.3.1.2. Socket types and protocols

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

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

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

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

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

Other socket types can be defined.

Each socket may have a specific *protocol* associated with it. This protocol is used within the domain to provide the semantics required by the socket type. Not all socket types are supported by each domain; support depends on the existence and the implementation of a suitable protocol within the domain. For example, within the "Internet" domain, the SOCK_DGRAM type may be implemented by the UDP user datagram protocol, and the SOCK_STREAM type may be implemented by the TCP transmission control protocol, while no standard protocols to provide SOCK_RDM or SOCK_SEQPACKET sockets exist.

2.3.1.3. Socket creation, naming and service establishment

Sockets may be *connected* or *unconnected*. An unconnected socket descriptor is obtained by the *socket* call:

```
s = socket(domain, type, protocol);
result int s; int domain, type, protocol;
```

The socket domain and type are as described above, and are specified using the definitions from *<sys/socket.h>*. The protocol may be given as 0, meaning any suitable protocol. One of several possible protocols may be selected using identifiers obtained from a library routine, *getprotobyname*.

An unconnected socket descriptor of a connection-oriented type may yield a connected socket descriptor in one of two ways: either by actively connecting to another socket, or by becoming associated with a name in the communications domain and *accepting* a connection from another socket. Datagram sockets need not establish connections before use.

To accept connections or to receive datagrams, a socket must first have a binding to a name (or address) within the communications domain. Such a binding may be established by a *bind* call:

```
bind(s, name, namelen);
int s; struct sockaddr *name; int namelen;
```

Datagram sockets may have default bindings established when first sending data if not explicitly bound earlier. In either case, a socket's bound name may be retrieved with a *getsockname* call:

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

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

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

Domains may support sockets with several names.

2.3.1.4. Accepting connections

Once a binding is made to a connection-oriented socket, it is possible to *listen* for connections:

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

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

An *accept* call:

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

returns a descriptor for a new, connected, socket from the queue of pending connections on *s*. If no new connections are queued for acceptance, the call will wait for a connection unless non-blocking I/O has been enabled.

2.3.1.5. Making connections

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

```
connect(s, name, namelen);
int s; struct sockaddr *name; int namelen;
```

Although datagram sockets do not establish connections, the *connect* call may be used with such sockets to create an *association* with the foreign address. The address is recorded for use in future *send* calls, which then need not supply destination addresses. Datagrams will be received only from that peer, and asynchronous error reports may be received.

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

† 4.3BSD supports *socketpair* creation only in the "unix" communication domain.

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

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

The call

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

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

2.3.1.6. Sending and receiving data

Messages may be sent from a socket by:

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

if the socket is not connected or:

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

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

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

and

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

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

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

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

2.3.1.7. Scatter/gather and exchanging access rights

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

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

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

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

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

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

2.3.1.8. Using read and write with sockets

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

2.3.1.9. Shutting down halves of full-duplex connections

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

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

where *direction* is 0 to not read further, 1 to not write further, or 2 to completely shut the connection down. If the underlying protocol supports unidirectional or bidirectional shutdown, this indication will be passed to the peer. For example, a shutdown for writing might produce an end-of-file condition at the remote end.

2.3.1.10. Socket and protocol options

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

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

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

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

2.3.2. UNIX domain

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

2.3.2.1. Types of sockets

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

2.3.2.2. Naming

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

2.3.2.3. Access rights transmission

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

2.3.3. INTERNET domain

This section describes briefly how the Internet domain is mapped to the model described in this section. More information will be found in the document describing the network implementation in 4.3BSD.

2.3.3.1. Socket types and protocols

SOCK_STREAM is supported by the Internet TCP protocol; SOCK_DGRAM by the UDP protocol. Each is layered atop the transport-level Internet Protocol (IP). The Internet Control Message Protocol is implemented atop/beside IP and is accessible via a raw socket. The SOCK_SEQPACKET has no direct Internet family analogue; a protocol based on one from the XEROX NS family and layered on top of IP could be implemented to fill this gap.

2.3.3.2. Socket naming

Sockets in the Internet domain have names composed of the 32 bit Internet address, and a 16 bit port number. Options may be used to provide IP source routing or security options. The 32-bit address is composed of network and host parts; the network part is variable in size and is frequency encoded. The host part may optionally be interpreted as a subnet field plus the host on subnet; this is is enabled by setting a network address mask at boot time.

2.3.3.3. Access rights transmission

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

2.3.3.4. Raw access

The Internet domain allows the super-user access to the raw facilities of IP. These interfaces are modeled as SOCK_RAW sockets. Each raw socket is associated with one IP protocol number, and receives all traffic received for that protocol. This allows administrative and debugging functions to occur, and enables user-level implementations of special-purpose protocols such as inter-gateway routing protocols.

† The 4.3BSD implementation of the UNIX domain embeds bound sockets in the UNIX file system name space; this may change in future releases.

2.4. Terminals and Devices

2.4.1. Terminals

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

2.4.1.1. Terminal input

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

2.4.1.1.1. Input modes

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

2.4.1.1.2. Interrupt characters

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

2.4.1.1.3. Line editing

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

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

Input characters may be echoed to the terminal as they are received. Non-graphic ASCII input characters may be echoed as a two-character printable representation, "^character."

PS 6 1

2.4.1.2. Terminal output

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

2.4.1.3. Terminal control operations

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

† The control interface described here is an internal interface only in 4.3BSD. Future releases will probably use a modified interface based on currently-proposed standards.

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

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

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

2.4.1.4. Terminal hardware support

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

2.4.2. Structured devices

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

2.4.3. Unstructured devices

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

2.5. Process and kernel descriptors

The status of the facilities in this section is still under discussion. The *ptrace* facility of earlier UNIX systems remains in 4.3BSD. Planned enhancements would allow a descriptor-based process control facility.

I. Summary of facilities

1. Kernel primitives

1.1. Process naming and protection

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

1.2 Memory management

<sys/mman.h>	memory management definitions
sbrk	change data section size
sstk†	change stack section size
getpagesize	get memory page size
mmap†	map pages of memory
msync†	flush modified mapped pages to filesystem
munmap†	unmap memory
mprotect†	change protection of pages
madvise†	give memory management advice
mincore†	determine core residency of pages
msleep†	sleep on a lock
mwakeup†	wakeup process sleeping on a lock

1.3 Signals

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

1.4 Timing and statistics

<sys/time.h>	time-related definitions
gettimeofday	get current time and timezone
settimeofday	set current time and timezone

† Not supported in 4.3BSD.

getitimer	read an interval timer
setitimer	get and set an interval timer
profil	profile process

1.5 Descriptors

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

1.6 Resource controls

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

1.7 System operation support

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

2. System facilities

2.1 Generic operations

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

2.2 File system

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

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

† Not supported in 4.3BSD.

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

2.3 Communications

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

2.4 Terminals, block and character devices

2.5 Processes and kernel hooks

An Introductory 4.3BSD Interprocess Communication Tutorial

Stuart Sechrest

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

ABSTRACT

Berkeley UNIX† 4.3BSD offers several choices for interprocess communication. To aid the programmer in developing programs which are comprised of cooperating processes, the different choices are discussed and a series of example programs are presented. These programs demonstrate in a simple way the use of pipes, socketpairs, sockets and the use of datagram and stream communication. The intent of this document is to present a few simple example programs, not to describe the networking system in full.

1. Goals

Facilities for interprocess communication (IPC) and networking were a major addition to UNIX in the Berkeley UNIX 4.2BSD release. These facilities required major additions and some changes to the system interface. The basic idea of this interface is to make IPC similar to file I/O. In UNIX a process has a set of I/O descriptors, from which one reads and to which one writes. Descriptors may refer to normal files, to devices (including terminals), or to communication channels. The use of a descriptor has three phases: its creation, its use for reading and writing, and its destruction. By using descriptors to write files, rather than simply naming the target file in the write call, one gains a surprising amount of flexibility. Often, the program that creates a descriptor will be different from the program that uses the descriptor. For example the shell can create a descriptor for the output of the 'ls' command that will cause the listing to appear in a file rather than on a terminal. Pipes are another form of descriptor that have been used in UNIX for some time. Pipes allow one-way data transmission from one process to another; the two processes and the pipe must be set up by a common ancestor.

The use of descriptors is not the only communication interface provided by UNIX. The signal mechanism sends a tiny amount of information from one process to another. The signaled process receives only the signal type, not the identity of the sender, and the number of possible signals is small. The signal semantics limit the flexibility of the signaling mechanism as a means of interprocess communication.

The identification of IPC with I/O is quite longstanding in UNIX and has proved quite successful. At first, however, IPC was limited to processes communicating within a single machine. With Berkeley UNIX 4.2BSD this expanded to include IPC between machines. This expansion has necessitated some change in the way that descriptors are created.

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

Additionally, new possibilities for the meaning of read and write have been admitted. Originally the meanings, or semantics, of these terms were fairly simple. When you wrote something it was delivered. When you read something, you were blocked until the data arrived. Other possibilities exist, however. One can write without full assurance of delivery if one can check later to catch occasional failures. Messages can be kept as discrete units or merged into a stream. One can ask to read, but insist on not waiting if nothing is immediately available. These new possibilities are allowed in the Berkeley UNIX IPC interface.

Thus Berkeley UNIX 4.3BSD offers several choices for IPC. This paper presents simple examples that illustrate some of the choices. The reader is presumed to be familiar with the C programming language [Kernighan & Ritchie 1978], but not necessarily with the system calls of the UNIX system or with processes and interprocess communication. The paper reviews the notion of a process and the types of communication that are supported by Berkeley UNIX 4.3BSD. A series of examples are presented that create processes that communicate with one another. The programs show different ways of establishing channels of communication. Finally, the calls that actually transfer data are reviewed. To clearly present how communication can take place, the example programs have been cleared of anything that might be construed as useful work. They can, therefore, serve as models for the programmer trying to construct programs which are comprised of cooperating processes.

2. Processes

A *program* is both a sequence of statements and a rough way of referring to the computation that occurs when the compiled statements are run. A *process* can be thought of as a single line of control in a program. Most programs execute some statements, go through a few loops, branch in various directions and then end. These are single process programs. Programs can also have a point where control splits into two independent lines, an action called *forking.* In UNIX these lines can never join again. A call to the system routine *fork()*, causes a process to split in this way. The result of this call is that two independent processes will be running, executing exactly the same code. Memory values will be the same for all values set before the fork, but, subsequently, each version will be able to change only the value of its own copy of each variable. Initially, the only difference between the two will be the value returned by *fork().* The parent will receive a process id for the child, the child will receive a zero. Calls to *fork(),* therefore, typically precede, or are included in, an if-statement.

A process views the rest of the system through a private table of descriptors. The descriptors can represent open files or sockets (sockets are communication objects that will be discussed below). Descriptors are referred to by their index numbers in the table. The first three descriptors are often known by special names, *stdin, stdout* and *stderr.* These are the standard input, output and error. When a process forks, its descriptor table is copied to the child. Thus, if the parent's standard input is being taken from a terminal (devices are also treated as files in UNIX), the child's input will be taken from the same terminal. Whoever reads first will get the input. If, before forking, the parent changes its standard input so that it is reading from a new file, the child will take its input from the new file. It is also possible to take input from a socket, rather than from a file.

3. Pipes

Most users of UNIX know that they can pipe the output of a program "prog1" to the input of another, "prog2," by typing the command *"prog1 | prog2."* This is called "piping" the output of one program to another because the mechanism used to transfer the output is called a pipe. When the user types a command, the command is read by the shell, which decides how to execute it. If the command is simple, for example, *"prog1,"* the shell forks a process, which executes the program, prog1, and then dies. The shell waits for this termination and then prompts for the next command. If the command is a compound command, *"prog1 | prog2,"* the shell creates two processes connected by a pipe. One process runs the

program, prog1, the other runs prog2. The pipe is an I/O mechanism with two ends, or sockets. Data that is written into one socket can be read from the other.

Since a program specifies its input and output only by the descriptor table indices, which appear as variables or constants, the input source and output destination can be changed without changing the text of the program. It is in this way that the shell is able to set up pipes. Before executing prog1, the process can close whatever is at *stdout* and replace it with one end of a pipe. Similarly, the process that will execute prog2 can substitute the opposite

```
#include <stdio.h>

#define DATA "Bright star, would I were steadfast as thou art . . ."

/*
 * This program creates a pipe, then forks.  The child communicates to the
 * parent over the pipe. Notice that a pipe is a one-way communications
 * device.  I can write to the output socket (sockets[1], the second socket
 * of the array returned by pipe()) and read from the input socket
 * (sockets[0]), but not vice versa.
 */

main()
{
      int sockets[2], child;

      /* Create a pipe */
      if (pipe(sockets) < 0) {
            perror("opening stream socket pair");
            exit(10);
      }

      if ((child = fork()) == -1)
            perror("fork");
      else if (child) {
            char buf[1024];

            /* This is still the parent.  It reads the child's message. */
            close(sockets[1]);
            if (read(sockets[0], buf, 1024) < 0)
                  perror("reading message");
            printf("-->%s\n", buf);
            close(sockets[0]);
      } else {
            /* This is the child.  It writes a message to its parent. */
            close(sockets[0]);
            if (write(sockets[1], DATA, sizeof(DATA)) < 0)
                  perror("writing message");
            close(sockets[1]);
      }
}
```

Figure 1 Use of a pipe

end of the pipe for *stdin.*

Let us now examine a program that creates a pipe for communication between its child and itself (Figure 1). A pipe is created by a parent process, which then forks. When a process forks, the parent's descriptor table is copied into the child's.

In Figure 1, the parent process makes a call to the system routine *pipe()*. This routine creates a pipe and places descriptors for the sockets for the two ends of the pipe in the process's descriptor table. *Pipe()* is passed an array into which it places the index numbers of the sockets it created. The two ends are not equivalent. The socket whose index is returned in the low word of the array is opened for reading only, while the socket in the high end is opened only for writing. This corresponds to the fact that the standard input is the first descriptor of a process's descriptor table and the standard output is the second. After creating the pipe, the parent creates the child with which it will share the pipe by calling *fork()*. Figure 2 illustrates the effect of a fork. The parent process's descriptor table points to both ends of the pipe. After the fork, both parent's and child's descriptor tables point to the pipe. The child can then use the pipe to send a message to the parent.

Just what is a pipe? It is a one-way communication mechanism, with one end opened for reading and the other end for writing. Therefore, parent and child need to agree on which way to turn the pipe, from parent to child or the other way around. Using the same pipe for communication both from parent to child and from child to parent would be possible (since both processes have references to both ends), but very complicated. If the parent and child are to have a two-way conversation, the parent creates two pipes, one for use in each direction. (In accordance with their plans, both parent and child in the example above close the socket that they will not use. It is not required that unused descriptors be closed, but it is good practice.) A pipe is also a *stream* communication mechanism; that is, all messages sent through the pipe are placed in order and reliably delivered. When the reader asks for a certain number of bytes from this stream, he is given as many bytes as are available, up to the amount of the request. Note that these bytes may have come from the same call to *write()* or from several calls to *write()* which were concatenated.

4. Socketpairs

Berkeley UNIX 4.3BSD provides a slight generalization of pipes. A pipe is a pair of connected sockets for one-way stream communication. One may obtain a pair of connected sockets for two-way stream communication by calling the routine *socketpair()*. The program in Figure 3 calls *socketpair()* to create such a connection. The program uses the link for communication in both directions. Since socketpairs are an extension of pipes, their use resembles that of pipes. Figure 4 illustrates the result of a fork following a call to *socketpair()*.

Socketpair() takes as arguments a specification of a domain, a style of communication, and a protocol. These are the parameters shown in the example. Domains and protocols will be discussed in the next section. Briefly, a domain is a space of names that may be bound to sockets and implies certain other conventions. Currently, socketpairs have only been implemented for one domain, called the UNIX domain. The UNIX domain uses UNIX path names for naming sockets. It only allows communication between sockets on the same machine.

Note that the header files *<sys/socket.h>* and *<sys/types.h>*. are required in this program. The constants AF_UNIX and SOCK_STREAM are defined in *<sys/socket.h>*, which in turn requires the file *<sys/types.h>* for some of its definitions.

5. Domains and Protocols

Pipes and socketpairs are a simple solution for communicating between a parent and child or between child processes. What if we wanted to have processes that have no common ancestor with whom to set up communication? Neither standard UNIX pipes nor socketpairs

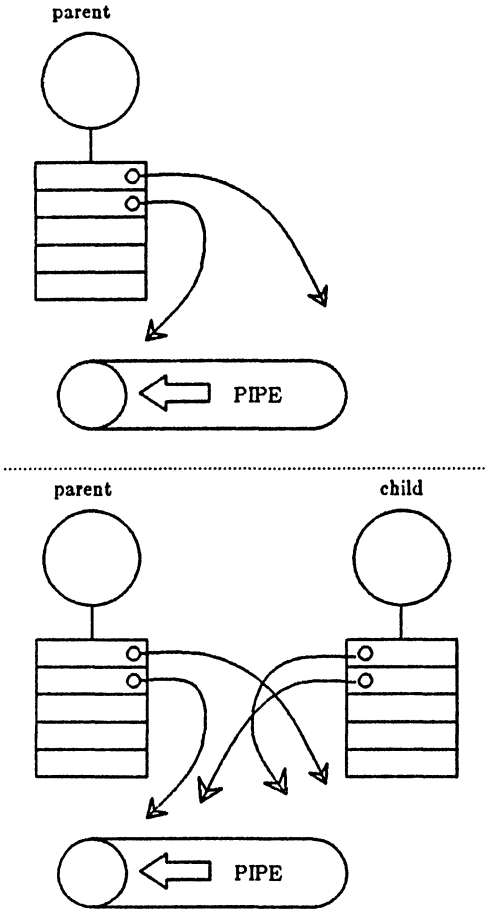


Figure 2 Sharing a pipe between parent and child

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>

#define DATA1 "In Xanadu, did Kublai Khan . . ."
#define DATA2 "A stately pleasure dome decree . . ."

/*
 * This program creates a pair of connected sockets then forks and
 * communicates over them.  This is very similar to communication with pipes
 * however, socketpairs are two-way communications objects. Therefore I can
 * send messages in both directions.
 */

main()
{
      int sockets[2], child;
      char buf[1024];

      if (socketpair(AF_UNIX, SOCK_STREAM, 0, sockets) < 0) {
            perror("opening stream socket pair");
            exit(1);
      }

      if ((child = fork()) == -1)
            perror("fork");
      else if (child) {/* This is the parent. */
            close(sockets[0]);
            if (read(sockets[1], buf, 1024, 0) < 0)
                  perror("reading stream message");
            printf("-->%s\n", buf);
            if (write(sockets[1], DATA2, sizeof(DATA2)) < 0)
                  perror("writing stream message");
            close(sockets[1]);
      } else {          /* This is the child. */
            close(sockets[1]);
            if (write(sockets[0], DATA1, sizeof(DATA1)) < 0)
                  perror("writing stream message");
            if (read(sockets[0], buf, 1024, 0) < 0)
                  perror("reading stream message");
            printf("-->%s\n", buf);
            close(sockets[0]);
      }
}
```

Figure 3 Use of a socketpair

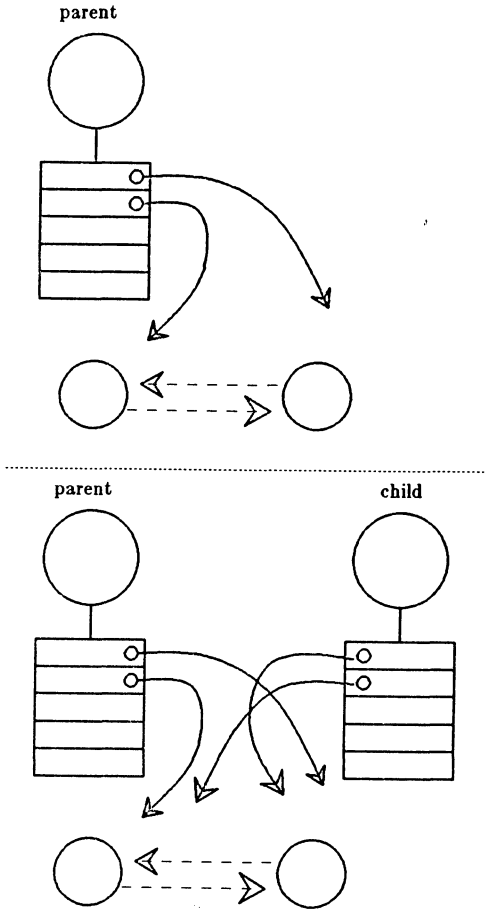


Figure 4 Sharing a socketpair between parent and child

are the answer here, since both mechanisms require a common ancestor to set up the communication. We would like to have two processes separately create sockets and then have messages sent between them. This is often the case when providing or using a service in the system. This is also the case when the communicating processes are on separate machines. In Berkeley UNIX 4.3BSD one can create individual sockets, give them names and send messages between them.

Sockets created by different programs use names to refer to one another; names generally must be translated into addresses for use. The space from which an address is drawn is referred to as a *domain.* There are several domains for sockets. Two that will be used in the examples here are the UNIX domain (or AF_UNIX, for Address Format UNIX) and the Internet domain (or AF_INET). UNIX domain IPC is an experimental facility in 4.2BSD and 4.3BSD. In the UNIX domain, a socket is given a path name within the file system name space. A file system node is created for the socket and other processes may then refer to the

socket by giving the proper pathname. UNIX domain names, therefore, allow communication between any two processes that work in the same file system. The Internet domain is the UNIX implementation of the DARPA Internet standard protocols IP/TCP/UDP. Addresses in the Internet domain consist of a machine network address and an identifying number, called a port. Internet domain names allow communication between machines.

Communication follows some particular "style." Currently, communication is either through a *stream* or by *datagram.* Stream communication implies several things. Communication takes place across a connection between two sockets. The communication is reliable, error-free, and, as in pipes, no message boundaries are kept. Reading from a stream may result in reading the data sent from one or several calls to *write()* or only part of the data from a single call, if there is not enough room for the entire message, or if not all the data from a large message has been transferred. The protocol implementing such a style will retransmit messages received with errors. It will also return error messages if one tries to send a message after the connection has been broken. Datagram communication does not use connections. Each message is addressed individually. If the address is correct, it will generally be received, although this is not guaranteed. Often datagrams are used for requests that require a response from the recipient. If no response arrives in a reasonable amount of time, the request is repeated. The individual datagrams will be kept separate when they are read, that is, message boundaries are preserved.

The difference in performance between the two styles of communication is generally less important than the difference in semantics. The performance gain that one might find in using datagrams must be weighed against the increased complexity of the program, which must now concern itself with lost or out of order messages. If lost messages may simply be ignored, the quantity of traffic may be a consideration. The expense of setting up a connection is best justified by frequent use of the connection. Since the performance of a protocol changes as it is tuned for different situations, it is best to seek the most up-to-date information when making choices for a program in which performance is crucial.

A protocol is a set of rules, data formats and conventions that regulate the transfer of data between participants in the communication. In general, there is one protocol for each socket type (stream, datagram, etc.) within each domain. The code that implements a protocol keeps track of the names that are bound to sockets, sets up connections and transfers data between sockets, perhaps sending the data across a network. This code also keeps track of the names that are bound to sockets. It is possible for several protocols, differing only in low level details, to implement the same style of communication within a particular domain. Although it is possible to select which protocol should be used, for nearly all uses it is sufficient to request the default protocol. This has been done in all of the example programs.

One specifies the domain, style and protocol of a socket when it is created. For example, in Figure 5a the call to *socket()* causes the creation of a datagram socket with the default protocol in the UNIX domain.

6. Datagrams in the UNIX Domain

Let us now look at two programs that create sockets separately. The programs in Figures 5a and 5b use datagram communication rather than a stream. The structure used to name UNIX domain sockets is defined in the file *<sys/un.h>*. The definition has also been included in the example for clarity.

Each program creates a socket with a call to *socket().* These sockets are in the UNIX domain. Once a name has been decided upon it is attached to a socket by the system call *bind().* The program in Figure 5a uses the name "socket", which it binds to its socket. This name will appear in the working directory of the program. The routines in Figure 5b use its socket only for sending messages. It does not create a name for the socket because no other process has to refer to it.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

/*
 * In the included file <sys/un.h> a sockaddr_un is defined as follows
 * struct sockaddr_un {
 *    short sun_family;
 *    char  sun_path[108];
 * };
 */

#include <stdio.h>

#define NAME "socket"

/*
 * This program creates a UNIX domain datagram socket, binds a name to it,
 * then reads from the socket.
 */
main()
{
      int sock, length;
      struct sockaddr_un name;
      char buf[1024];

      /* Create socket from which to read. */
      sock = socket(AF_UNIX, SOCK_DGRAM, 0);
      if (sock < 0) {
            perror("opening datagram socket");
            exit(1);
      }
      /* Create name. */
      name.sun_family = AF_UNIX;
      strcpy(name.sun_path, NAME);
      if (bind(sock, &name, sizeof(struct sockaddr_un))) {
            perror("binding name to datagram socket");
            exit(1);
      }
      printf("socket -->%s\n", NAME);
      /* Read from the socket */
      if (read(sock, buf, 1024) < 0)
            perror("receiving datagram packet");
      printf("-->%s\n", buf);
      close(sock);
      unlink(NAME);
}
```

Figure 5a Reading UNIX domain datagrams

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define DATA "The sea is calm tonight, the tide is full . . ."

/*
 * Here I send a datagram to a receiver whose name I get from the command
 * line arguments.  The form of the command line is udgramsend pathname
 */

main(argc, argv)
      int argc;
      char *argv[];
{
      int sock;
      struct sockaddr_un name;

      /* Create socket on which to send. */
      sock = socket(AF_UNIX, SOCK_DGRAM, 0);
      if (sock < 0) {
            perror("opening datagram socket");
            exit(1);
      }
      /* Construct name of socket to send to. */
      name.sun_family = AF_UNIX;
      strcpy(name.sun_path, argv[1]);
      /* Send message. */
      if (sendto(sock, DATA, sizeof(DATA), 0,
          &name, sizeof(struct sockaddr_un)) < 0) {
            perror("sending datagram message");
      }
      close(sock);
}
```

Figure 5b Sending a UNIX domain datagrams

Names in the UNIX domain are path names. Like file path names they may be either absolute (e.g. "/dev/imaginary") or relative (e.g. "socket"). Because these names are used to allow processes to rendezvous, relative path names can pose difficulties and should be used with care. When a name is bound into the name space, a file (inode) is allocated in the file system. If the inode is not deallocated, the name will continue to exist even after the bound socket is closed. This can cause subsequent runs of a program to find that a name is unavailable, and can cause directories to fill up with these objects. The names are removed by calling *unlink()* or using the *rm*(1) command. Names in the UNIX domain are only used for rendezvous. They are not used for message delivery once a connection is established. Therefore, in contrast with the Internet domain, unbound sockets need not be (and are not) automatically given addresses when they are connected.

There is no established means of communicating names to interested parties. In the example, the program in Figure 5b gets the name of the socket to which it will send its message through its command line arguments. Once a line of communication has been created, one can send the names of additional, perhaps new, sockets over the link. Facilities will have to be built that will make the distribution of names less of a problem than it now is.

7. Datagrams in the Internet Domain

The examples in Figure 6a and 6b are very close to the previous example except that the socket is in the Internet domain. The structure of Internet domain addresses is defined in the file *<netinet/in.h>*. Internet addresses specify a host address (a 32-bit number) and a delivery slot, or port, on that machine. These ports are managed by the system routines that implement a particular protocol. Unlike UNIX domain names, Internet socket names are not entered into the file system and, therefore, they do not have to be unlinked after the socket has been closed. When a message must be sent between machines it is sent to the protocol routine on the destination machine, which interprets the address to determine to which socket the message should be delivered. Several different protocols may be active on the same machine, but, in general, they will not communicate with one another. As a result, different protocols are allowed to use the same port numbers. Thus, implicitly, an Internet address is a triple including a protocol as well as the port and machine address. An *association* is a temporary or permanent specification of a pair of communicating sockets. An association is thus identified by the tuple *<protocol, local machine address, local port, remote machine address, remote port>*. An association may be transient when using datagram sockets; the association actually exists during a *send* operation.

The protocol for a socket is chosen when the socket is created. The local machine address for a socket can be any valid network address of the machine, if it has more than one, or it can be the wildcard value INADDR_ANY. The wildcard value is used in the program in Figure 6a. If a machine has several network addresses, it is likely that messages sent to any of the addresses should be deliverable to a socket. This will be the case if the wildcard value has been chosen. Note that even if the wildcard value is chosen, a program sending messages to the named socket must specify a valid network address. One can be willing to receive from "anywhere," but one cannot send a message "anywhere." The program in Figure 6b is given the destination host name as a command line argument. To determine a network address to which it can send the message, it looks up the host address by the call to *gethostbyname()*. The returned structure includes the host's network address, which is copied into the structure specifying the destination of the message.

The port number can be thought of as the number of a mailbox, into which the protocol places one's messages. Certain daemons, offering certain advertised services, have reserved or "well-known" port numbers. These fall in the range from 1 to 1023. Higher numbers are available to general users. Only servers need to ask for a particular number. The system will assign an unused port number when an address is bound to a socket. This may happen when an explicit *bind* call is made with a port number of 0, or when a *connect* or *send* is performed on an unbound socket. Note that port numbers are not automatically reported back to the user. After calling *bind()*, asking for port 0, one may call *getsockname()* to discover what port was actually assigned. The routine *getsockname()* will not work for names in the UNIX domain.

The format of the socket address is specified in part by standards within the Internet domain. The specification includes the order of the bytes in the address. Because machines differ in the internal representation they ordinarily use to represent integers, printing out the port number as returned by *getsockname()* may result in a misinterpretation. To print out the number, it is necessary to use the routine *ntohs()* (for *network to host: short*) to convert the number from the network representation to the host's representation. On some machines, such as 68000-based machines, this is a null operation. On others, such as VAXes, this results in a swapping of bytes. Another routine exists to convert a short integer from the host format to the network format, called *htons()*; similar routines exist for long integers. For further information, refer to the entry for *byteorder* in section 3 of the manual.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>

/*
 * In the included file <netinet/in.h> a sockaddr_in is defined as follows:
 * struct sockaddr_in {
 *    short sin_family;
 *    u_short     sin_port;
 *    struct in_addr sin_addr;
 *    char  sin_zero[8];
 * };
 *
 * This program creates a datagram socket, binds a name to it, then reads
 * from the socket.
 */
main()
{
      int sock, length;
      struct sockaddr_in name;
      char buf[1024];

      /* Create socket from which to read. */
      sock = socket(AF_INET, SOCK_DGRAM, 0);
      if (sock < 0) {
            perror("opening datagram socket");
            exit(1);
      }
      /* Create name with wildcards. */
      name.sin_family = AF_INET;
      name.sin_addr.s_addr = INADDR_ANY;
      name.sin_port = 0;
      if (bind(sock, &name, sizeof(name))) {
            perror("binding datagram socket");
            exit(1);
      }
      /* Find assigned port value and print it out. */
      length = sizeof(name);
      if (getsockname(sock, &name, &length)) {
            perror("getting socket name");
            exit(1);
      }
      printf("Socket has port #%d\n", ntohs(name.sin_port));
      /* Read from the socket */
      if (read(sock, buf, 1024) < 0)
            perror("receiving datagram packet");
      printf("-->%s\n", buf);
      close(sock);
}
```

Figure 6a Reading Internet domain datagrams

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define DATA "The sea is calm tonight, the tide is full . . ."

/*
 * Here I send a datagram to a receiver whose name I get from the command
 * line arguments.  The form of the command line is dgramsend hostname
 * portnumber
 */

main(argc, argv)
      int argc;
      char *argv[];
{
      int sock;
      struct sockaddr_in name;
      struct hostent *hp, *gethostbyname();

      /* Create socket on which to send. */
      sock = socket(AF_INET, SOCK_DGRAM, 0);
      if (sock < 0) {
            perror("opening datagram socket");
            exit(1);
      }
      /*
       * Construct name, with no wildcards, of the socket to send to.
       * Getnostbyname() returns a structure including the network address
       * of the specified host.  The port number is taken from the command
       * line.
       */
      hp = gethostbyname(argv[1]);
      if (hp == 0) {
            fprintf(stderr, "%s: unknown host0, argv[1]);
            exit(2);
      }
      bcopy(hp->h_addr, &name.sin_addr, hp->h_length);
      name.sin_family = AF_INET;
      name.sin_port = htons(atoi(argv[2]));
      /* Send message. */
      if (sendto(sock, DATA, sizeof(DATA), 0, &name, sizeof(name)) < 0)
            perror("sending datagram message");
      close(sock);
}
```

Figure 6b Sending an Internet domain datagram

8. Connections

To send data between stream sockets (having communication style SOCK_STREAM), the sockets must be connected. Figures 7a and 7b show two programs that create such a connection. The program in 7a is relatively simple. To initiate a connection, this program

simply creates a stream socket, then calls *connect()*, specifying the address of the socket to which it wishes its socket connected. Provided that the target socket exists and is prepared to handle a connection, connection will be complete, and the program can begin to send messages. Messages will be delivered in order without message boundaries, as with pipes. The connection is destroyed when either socket is closed (or soon thereafter). If a process persists in sending messages after the connection is closed, a SIGPIPE signal is sent to the process by the operating system. Unless explicit action is taken to handle the signal (see the manual page for *signal* or *sigvec*), the process will terminate and the shell will print the message "broken pipe."

Forming a connection is asymmetrical; one process, such as the program in Figure 7a, requests a connection with a particular socket, the other process accepts connection requests. Before a connection can be accepted a socket must be created and an address bound to it. This situation is illustrated in the top half of Figure 8. Process 2 has created a socket and bound a port number to it. Process 1 has created an unnamed socket. The address bound to process 2's socket is then made known to process 1 and, perhaps to several other potential communicants as well. If there are several possible communicants, this one socket might receive several requests for connections. As a result, a new socket is created for each connection. This new socket is the endpoint for communication within this process for this connection. A connection may be destroyed by closing the corresponding socket.

The program in Figure 7b is a rather trivial example of a server. It creates a socket to which it binds a name, which it then advertises. (In this case it prints out the socket number.) The program then calls *listen()* for this socket. Since several clients may attempt to connect more or less simultaneously, a queue of pending connections is maintained in the system address space. *Listen()* marks the socket as willing to accept connections and initializes the queue. When a connection is requested, it is listed in the queue. If the queue is full, an error status may be returned to the requester. The maximum length of this queue is specified by the second argument of *listen()*; the maximum length is limited by the system. Once the listen call has been completed, the program enters an infinite loop. On each pass through the loop, a new connection is accepted and removed from the queue, and, hence, a new socket for the connection is created. The bottom half of Figure 8 shows the result of Process 1 connecting with the named socket of Process 2, and Process 2 accepting the connection. After the connection is created, the service, in this case printing out the messages, is performed and the connection socket closed. The *accept()* call will take a pending connection request from the queue if one is available, or block waiting for a request. Messages are read from the connection socket. Reads from an active connection will normally block until data is available. The number of bytes read is returned. When a connection is destroyed, the read call returns immediately. The number of bytes returned will be zero.

The program in Figure 7c is a slight variation on the server in Figure 7b. It avoids blocking when there are no pending connection requests by calling *select()* to check for pending requests before calling *accept()*. This strategy is useful when connections may be received on more than one socket, or when data may arrive on other connected sockets before another connection request.

The programs in Figures 9a and 9b show a program using stream communication in the UNIX domain. Streams in the UNIX domain can be used for this sort of program in exactly the same way as Internet domain streams, except for the form of the names and the restriction of the connections to a single file system. There are some differences, however, in the functionality of streams in the two domains, notably in the handling of *out-of-band* data (discussed briefly below). These differences are beyond the scope of this paper.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define DATA "Half a league, half a league . . ."

/*
 * This program creates a socket and initiates a connection with the socket
 * given in the command line.  One message is sent over the connection and
 * then the socket is closed, ending the connection. The form of the command
 * line is streamwrite hostname portnumber
 */

main(argc, argv)
      int argc;
      char *argv[];
{
      int sock;
      struct sockaddr_in server;
      struct hostent *hp, *gethostbyname();
      char buf[1024];

      /* Create socket */
      sock = socket(AF_INET, SOCK_STREAM, 0);
      if (sock < 0) {
            perror("opening stream socket");
            exit(1);
      }
      /* Connect socket using name specified by command line. */
      server.sin_family = AF_INET;
      hp = gethostbyname(argv[1]);
      if (hp == 0) {
            fprintf(stderr, "%s: unknown host0, argv[1]);
            exit(2);
      }
      bcopy(hp->h_addr, &server.sin_addr, hp->h_length);
      server.sin_port = htons(atoi(argv[2]));

      if (connect(sock, &server, sizeof(server)) < 0) {
            perror("connecting stream socket");
            exit(1);
      }
      if (write(sock, DATA, sizeof(DATA)) < 0)
            perror("writing on stream socket");
      close(sock);
}
```

Figure 7a Initiating an Internet domain stream connection

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1

/*
 * This program creates a socket and then begins an infinite loop. Each time
 * through the loop it accepts a connection and prints out messages from it.
 * When the connection breaks, or a termination message comes through, the
 * program accepts a new connection.
 */

main()
{
      int sock, length;
      struct sockaddr_in server;
      int msgsock;
      char buf[1024];
      int rval;
      int i;

      /* Create socket */
      sock = socket(AF_INET, SOCK_STREAM, 0);
      if (sock < 0) {
            perror("opening stream socket");
            exit(1);
      }
      /* Name socket using wildcards */
      server.sin_family = AF_INET;
      server.sin_addr.s_addr = INADDR_ANY;
      server.sin_port = 0;
      if (bind(sock, &server, sizeof(server))) {
            perror("binding stream socket");
            exit(1);
      }
      /* Find out assigned port number and print it out */
      length = sizeof(server);
      if (getsockname(sock, &server, &length)) {
            perror("getting socket name");
            exit(1);
      }
      printf("Socket has port #%d\n", ntohs(server.sin_port));

      /* Start accepting connections */
      listen(sock, 5);
      do {
            msgsock = accept(sock, 0, 0);
            if (msgsock == -1)
                  perror("accept");
            else do {
                  bzero(buf, sizeof(buf));
```

```
                if ((rval = read(msgsock, buf, 1024)) < 0)
                        perror("reading stream message");
                i = 0;
                if (rval == 0)
                        printf("Ending connection\n");
                else
                        printf("-->%s\n", buf);
            } while (rval != 0);
            close(msgsock);
        } while (TRUE);
        /*
         * Since this program has an infinite loop, the socket "sock" is
         * never explicitly closed.  However, all sockets will be closed
         * automatically when a process is killed or terminates normally.
         */
}
```

Figure 7b Accepting an Internet domain stream connection

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1

/*
 * This program uses select() to check that someone is trying to connect
 * before calling accept().
 */

main()
{
        int sock, length;
        struct sockaddr_in server;
        int msgsock;
        char buf[1024];
        int rval;
        fd_set ready;
        struct timeval to;

        /* Create socket */
        sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0) {
                perror("opening stream socket");
                exit(1);
        }
        /* Name socket using wildcards */
        server.sin_family = AF_INET;
        server.sin_addr.s_addr = INADDR_ANY;
        server.sin_port = 0;
        if (bind(sock, &server, sizeof(server))) {
```

```
        perror("binding stream socket");
        exit(1);
    }
    /* Find out assigned port number and print it out */
    length = sizeof(server);
    if (getsockname(sock, &server, &length)) {
        perror("getting socket name");
        exit(1);
    }
    printf("Socket has port #%d\n", ntohs(server.sin_port));

    /* Start accepting connections */
    listen(sock, 5);
    do {
        FD_ZERO(&ready);
        FD_SET(sock, &ready);
        to.tv_sec = 5;
        if (select(sock + 1, &ready, 0, 0, &to) < 0) {
            perror("select");
            continue;
        }
        if (FD_ISSET(sock, &ready)) {
            msgsock = accept(sock, (struct sockaddr *)0, (int *)0);
            if (msgsock == -1)
                perror("accept");
            else do {
                bzero(buf, sizeof(buf));
                if ((rval = read(msgsock, buf, 1024)) < 0)
                    perror("reading stream message");
                else if (rval == 0)
                    printf("Ending connection\n");
                else
                    printf("-->%s\n", buf);
            } while (rval > 0);
            close(msgsock);
        } else
            printf("Do something else\n");
    } while (TRUE);
}
```

Figure 7c Using select() to check for pending connections

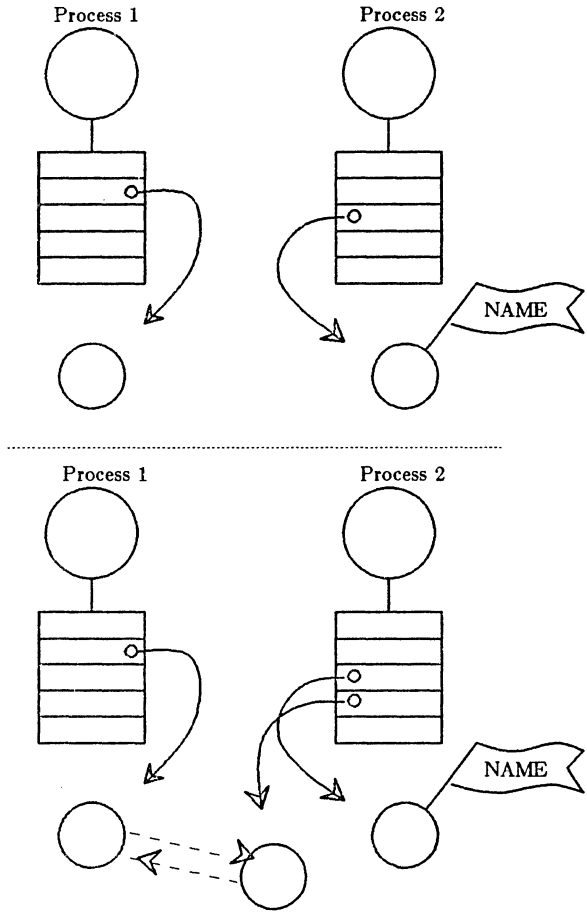


Figure 8 Establishing a stream connection

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define DATA "Half a league, half a league . . ."

/*
 * This program connects to the socket named in the command line and sends a
 * one line message to that socket. The form of the command line is
 * ustreamwrite pathname
 */
main(argc, argv)
      int argc;
      char *argv[];
{
      int sock;
      struct sockaddr_un server;
      char buf[1024];

      /* Create socket */
      sock = socket(AF_UNIX, SOCK_STREAM, 0);
      if (sock < 0) {
            perror("opening stream socket");
            exit(1);
      }
      /* Connect socket using name specified by command line. */
      server.sun_family = AF_UNIX;
      strcpy(server.sun_path, argv[1]);

      if (connect(sock, &server, sizeof(struct sockaddr_un)) < 0) {
            close(sock);
            perror("connecting stream socket");
            exit(1);
      }
      if (write(sock, DATA, sizeof(DATA)) < 0)
            perror("writing on stream socket");
}
```

Figure 9a Initiating a UNIX domain stream connection

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define NAME "socket"

/*
 * This program creates a socket in the UNIX domain and binds a name to it.
 * After printing the socket's name it begins a loop. Each time through the
 * loop it accepts a connection and prints out messages from it.  When the
 * connection breaks, or a termination message comes through, the program
```

```
 * accepts a new connection.
 */
main()
{
      int sock, msgsock, rval;
      struct sockaddr_un server;
      char buf[1024];

      /* Create socket */
      sock = socket(AF_UNIX, SOCK_STREAM, 0);
      if (sock < 0) {
            perror("opening stream socket");
            exit(1);
      }
      /* Name socket using file system name */
      server.sun_family = AF_UNIX;
      strcpy(server.sun_path, NAME);
      if (bind(sock, &server, sizeof(struct sockaddr_un))) {
            perror("binding stream socket");
            exit(1);
      }
      printf("Socket has name %s\n", server.sun_path);
      /* Start accepting connections */
      listen(sock, 5);
      for (;;) {
            msgsock = accept(sock, 0, 0);
            if (msgsock == -1)
                  perror("accept");
            else do {
                  bzero(buf, sizeof(buf));
                  if ((rval = read(msgsock, buf, 1024)) < 0)
                        perror("reading stream message");
                  else if (rval == 0)
                        printf("Ending connection\n");
                  else
                        printf("-->%s\n", buf);
            } while (rval > 0);
            close(msgsock);
      }
      /*
       * The following statements are not executed, because they follow an
       * infinite loop.  However, most ordinary programs will not run
       * forever.  In the UNIX domain it is necessary to tell the file
       * system that one is through using NAME.  In most programs one uses
       * the call unlink() as below. Since the user will have to kill this
       * program, it will be necessary to remove the name by a command from
       * the shell.
       */
      close(sock);
      unlink(NAME);
}
```

Figure 9b Accepting a UNIX domain stream connection

9. Reads, Writes, Recvs, etc.

UNIX 4.3BSD has several system calls for reading and writing information. The simplest calls are *read()* and *write()*. *Write()* takes as arguments the index of a descriptor, a pointer to a buffer containing the data and the size of the data. The descriptor may indicate either a file or a connected socket. "Connected" can mean either a connected stream socket (as described in Section 8) or a datagram socket for which a *connect()* call has provided a default destination (see the *connect()* manual page). *Read()* also takes a descriptor that indicates either a file or a socket. *Write()* requires a connected socket since no destination is specified in the parameters of the system call. *Read()* can be used for either a connected or an unconnected socket. These calls are, therefore, quite flexible and may be used to write applications that require no assumptions about the source of their input or the destination of their output. There are variations on *read()* and *write()* that allow the source and destination of the input and output to use several separate buffers, while retaining the flexibility to handle both files and sockets. These are *readv()* and *writev()*, for read and write *vector*.

It is sometimes necessary to send high priority data over a connection that may have unread low priority data at the other end. For example, a user interface process may be interpreting commands and sending them on to another process through a stream connection. The user interface may have filled the stream with as yet unprocessed requests when the user types a command to cancel all outstanding requests. Rather than have the high priority data wait to be processed after the low priority data, it is possible to send it as *out-of-band* (OOB) data. The notification of pending OOB data results in the generation of a SIGURG signal, if this signal has been enabled (see the manual page for *signal* or *sigvec*). See [Leffler 1986] for a more complete description of the OOB mechanism. There are a pair of calls similar to *read* and *write* that allow options, including sending and receiving OOB information; these are *send()* and *recv()*. These calls are used only with sockets; specifying a descriptor for a file will result in the return of an error status. These calls also allow *peeking* at data in a stream. That is, they allow a process to read data without removing the data from the stream. One use of this facility is to read ahead in a stream to determine the size of the next item to be read. When not using these options, these calls have the same functions as *read()* and *write()*.

To send datagrams, one must be allowed to specify the destination. The call *sendto()* takes a destination address as an argument and is therefore used for sending datagrams. The call *recvfrom()* is often used to read datagrams, since this call returns the address of the sender, if it is available, along with the data. If the identity of the sender does not matter, one may use *read()* or *recv()*.

Finally, there are a pair of calls that allow the sending and receiving of messages from multiple buffers, when the address of the recipient must be specified. These are *sendmsg()* and *recvmsg()*. These calls are actually quite general and have other uses, including, in the UNIX domain, the transmission of a file descriptor from one process to another.

The various options for reading and writing are shown in Figure 10, together with their parameters. The parameters for each system call reflect the differences in function of the different calls. In the examples given in this paper, the calls *read()* and *write()* have been used whenever possible.

10. Choices

This paper has presented examples of some of the forms of communication supported by Berkeley UNIX 4.3BSD. These have been presented in an order chosen for ease of presentation. It is useful to review these options emphasizing the factors that make each attractive.

Pipes have the advantage of portability, in that they are supported in all UNIX systems. They also are relatively simple to use. Socketpairs share this simplicity and have the additional advantage of allowing bidirectional communication. The major shortcoming of these mechanisms is that they require communicating processes to be descendants of a common

```
/*
 * The variable descriptor may be the descriptor of either a file
 * or of a socket.
 */
cc = read(descriptor, buf, nbytes)
int cc, descriptor; char *buf; int nbytes;

/*
 * An iovec can include several source buffers.
 */
cc = readv(descriptor, iov, iovcnt)
int cc, descriptor; struct iovec *iov; int iovcnt;

cc = write(descriptor, buf, nbytes)
int cc, descriptor; char *buf; int nbytes;

cc = writev(descriptor, iovec, ioveclen)
int cc, descriptor; struct iovec *iovec; int ioveclen;

/*
 * The variable ''sock'' must be the descriptor of a socket.
 * Flags may include MSG_OOB and MSG_PEEK.
 */
cc = send(sock, msg, len, flags)
int cc, sock; char *msg; int len, flags;

cc = sendto(sock, msg, len, flags, to, tolen)
int cc, sock; char *msg; int len, flags;
struct sockaddr *to; int tolen;

cc = sendmsg(sock, msg, flags)
int cc, sock; struct msghdr msg[]; int flags;

cc = recv(sock, buf, len, flags)
int cc, sock; char *buf; int len, flags;

cc = recvfrom(sock, buf, len, flags, from, fromlen)
int cc, sock; char *buf; int len, flags;
struct sockaddr *from; int *fromlen;

cc = recvmsg(sock, msg, flags)
int cc, socket; struct msghdr msg[]; int flags;
```

Figure 10 Varieties of read and write commands

process. They do not allow intermachine communication.

The two communication domains, UNIX and Internet, allow processes with no common ancestor to communicate. Of the two, only the Internet domain allows communication between machines. This makes the Internet domain a necessary choice for processes running on separate machines.

The choice between datagrams and stream communication is best made by carefully considering the semantic and performance requirements of the application. Streams can be both advantageous and disadvantageous. One disadvantage is that a process is only allowed a limited number of open streams, as there are usually only 64 entries available in the open descriptor table. This can cause problems if a single server must talk with a large number of clients. Another is that for delivering a short message the stream setup and teardown time can be unnecessarily long. Weighed against this are the reliability built into the streams. This will often be the deciding factor in favor of streams.

11. What to do Next

Many of the examples presented here can serve as models for multiprocess programs and for programs distributed across several machines. In developing a new multiprocess program, it is often easiest to first write the code to create the processes and communication paths. After this code is debugged, the code specific to the application can be added.

An introduction to the UNIX system and programming using UNIX system calls can be found in [Kernighan and Pike 1984]. Further documentation of the Berkeley UNIX 4.3BSD IPC mechanisms can be found in [Leffler et al. 1986]. More detailed information about particular calls and protocols is provided in sections 2, 3 and 4 of the UNIX Programmer's Manual [CSRG 1986]. In particular the following manual pages are relevant:

creating and naming sockets	socket(2), bind(2)
establishing connections	listen(2), accept(2), connect(2)
transferring data	read(2), write(2), send(2), recv(2)
addresses	inet(4F)
protocols	tcp(4P), udp(4P).

Acknowledgements

I would like to thank Sam Leffler and Mike Karels for their help in understanding the IPC mechanisms and all the people whose comments have helped in writing and improving this report.

This work was sponsored by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 4031, monitored by the Naval Electronics Systems Command under contract No. N00039-C-0235. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the US Government.

References

B.W. Kernighan & R. Pike, 1984,
The UNIX Programming Environment.
Englewood Cliffs, N.J.: Prentice-Hall.

B.W. Kernighan & D.M. Ritchie, 1978,
The C Programming Language,
Englewood Cliffs, N.J.: Prentice-Hall.

S.J. Leffler, R.S. Fabry, W.N. Joy, P. Lapsley, S. Miller & C. Torek, 1986,
An Advanced 4.3BSD Interprocess Communication Tutorial.
Computer Systems Research Group,
Department of Electrical Engineering and Computer Science,
University of California, Berkeley.

Computer Systems Research Group, 1986,
UNIX Programmer's Manual, 4.3 Berkeley Software Distribution.
Computer Systems Research Group,
Department of Electrical Engineering and Computer Science,
University of California, Berkeley.

An Advanced 4.3BSD Interprocess Communication Tutorial

Samuel J. Leffler

Robert S. Fabry

William N. Joy

Phil Lapsley

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

Steve Miller

Chris Torek

Heterogeneous Systems Laboratory
Department of Computer Science
University of Maryland, College Park
College Park, Maryland 20742

ABSTRACT

This document provides an introduction to the interprocess communication facilities included in the 4.3BSD release of the UNIX* system.

It discusses the overall model for interprocess communication and introduces the interprocess communication primitives which have been added to the system. The majority of the document considers the use of these primitives in developing applications. The reader is expected to be familiar with the C programming language as all examples are written in C.

* UNIX is a Trademark of Bell Laboratories.

1. INTRODUCTION

One of the most important additions to UNIX in 4.2BSD was interprocess communication. These facilities were the result of more than two years of discussion and research. The facilities provided in 4.2BSD incorporated many of the ideas from current research, while trying to maintain the UNIX philosophy of simplicity and conciseness. The current release of Berkeley UNIX, 4.3BSD, completes some of the IPC facilities and provides an upward-compatible interface. It is hoped that the interprocess communication facilities included in 4.3BSD will establish a standard for UNIX. From the response to the design, it appears many organizations carrying out work with UNIX are adopting it.

UNIX has previously been very weak in the area of interprocess communication. Prior to the 4BSD facilities, the only standard mechanism which allowed two processes to communicate were pipes (the mpx files which were part of Version 7 were experimental). Unfortunately, pipes are very restrictive in that the two communicating processes must be related through a common ancestor. Further, the semantics of pipes makes them almost impossible to maintain in a distributed environment.

Earlier attempts at extending the IPC facilities of UNIX have met with mixed reaction. The majority of the problems have been related to the fact that these facilities have been tied to the UNIX file system, either through naming or implementation. Consequently, the IPC facilities provided in 4.3BSD have been designed as a totally independent subsystem. The 4.3BSD IPC allows processes to rendezvous in many ways. Processes may rendezvous through a UNIX file system-like name space (a space where all names are path names) as well as through a network name space. In fact, new name spaces may be added at a future time with only minor changes visible to users. Further, the communication facilities have been extended to include more than the simple byte stream provided by a pipe. These extensions have resulted in a completely new part of the system which users will need time to familiarize themselves with. It is likely that as more use is made of these facilities they will be refined; only time will tell.

This document provides a high-level description of the IPC facilities in 4.3BSD and their use. It is designed to complement the manual pages for the IPC primitives by examples of their use. The remainder of this document is organized in four sections. Section 2 introduces the IPC-related system calls and the basic model of communication. Section 3 describes some of the supporting library routines users may find useful in constructing distributed applications. Section 4 is concerned with the client/server model used in developing applications and includes examples of the two major types of servers. Section 5 delves into advanced topics which sophisticated users are likely to encounter when using the IPC facilities.

2. BASICS

The basic building block for communication is the *socket*. A socket is an endpoint of communication to which a name may be *bound*. Each socket in use has a *type* and one or more associated processes. Sockets exist within *communication domains*. A communication domain is an abstraction introduced to bundle common properties of processes communicating through sockets. One such property is the scheme used to name sockets. For example, in the UNIX communication domain sockets are named with UNIX path names; e.g. a socket may be named "/dev/foo". Sockets normally exchange data only with sockets in the same domain (it may be possible to cross domain boundaries, but only if some translation process is performed). The 4.3BSD IPC facilities support three separate communication domains: the UNIX domain, for on-system communication; the Internet domain, which is used by processes which communicate using the the DARPA standard communication protocols; and the NS domain, which is used by processes which communicate using the Xerox standard communication protocols*. The underlying communication facilities provided by these domains have a significant influence on the internal system implementation as well as the interface to socket facilities available to a user. An example of the latter is that a socket "operating" in the UNIX domain sees a subset of the error conditions which are possible when operating in the Internet (or NS) domain.

2.1. Socket types

Sockets are typed according to the communication properties visible to a user. Processes are presumed to communicate only between sockets of the same type, although there is nothing that prevents communication between sockets of different types should the underlying communication protocols support this.

Four types of sockets currently are available to a user. A *stream* socket provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. Aside from the bidirectionality of data flow, a pair of connected stream sockets provides an interface nearly identical to that of pipes†.

A *datagram* socket supports bidirectional flow of data which is not promised to be sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket may find messages duplicated, and, possibly, in an order different from the order in which it was sent. An important characteristic of a datagram socket is that record boundaries in data are preserved. Datagram sockets closely model the facilities found in many contemporary packet switched networks such as the Ethernet.

A *raw* socket provides users access to the underlying communication protocols which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more esoteric facilities of an existing protocol. The use of raw sockets is considered in section 5.

A *sequenced packet* socket is similar to a stream socket, with the exception that record boundaries are preserved. This interface is provided only as part of the NS socket abstraction, and is very important in most serious NS applications. Sequenced-packet sockets allow the user to manipulate the SPP or IDP headers on a packet or a group of packets either by writing a prototype header along with whatever data is to be sent, or by specifying a default header to be used with all outgoing data, and allows the user to receive the headers on incoming packets. The use of these options is considered in section 5.

---

* See *Internet Transport Protocols*, Xerox System Integration Standard (XSIS)028112 for more information. This document is almost a necessity for one trying to write NS applications.

† In the UNIX domain, in fact, the semantics are identical and, as one might expect, pipes have been implemented internally as simply a pair of connected stream sockets.

Another potential socket type which has interesting properties is the *reliably delivered message* socket. The reliably delivered message socket has similar properties to a datagram socket, but with reliable delivery. There is currently no support for this type of socket, but a reliably delivered message protocol similar to Xerox's Packet Exchange Protocol (PEX) may be simulated at the user level. More information on this topic can be found in section 5.

2.2. Socket creation

To create a socket the *socket* system call is used:

```
s = socket(domain, type, protocol);
```

This call requests that the system create a socket in the specified *domain* and of the specified *type.* A particular protocol may also be requested. If the protocol is left unspecified (a value of 0), the system will select an appropriate protocol from those protocols which comprise the communication domain and which may be used to support the requested socket type. The user is returned a descriptor (a small integer number) which may be used in later system calls which operate on sockets. The domain is specified as one of the manifest constants defined in the file *<sys/socket.h>*. For the UNIX domain the constant is AF_UNIX*; for the Internet domain AF_INET; and for the NS domain, AF_NS. The socket types are also defined in this file and one of SOCK_STREAM, SOCK_DGRAM, SOCK_RAW, or SOCK_SEQPACKET must be specified. To create a stream socket in the Internet domain the following call might be used:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

This call would result in a stream socket being created with the TCP protocol providing the underlying communication support. To create a datagram socket for on-machine use the call might be:

```
s = socket(AF_UNIX, SOCK_DGRAM, 0);
```

The default protocol (used when the *protocol* argument to the *socket* call is 0) should be correct for most every situation. However, it is possible to specify a protocol other than the default; this will be covered in section 5.

There are several reasons a socket call may fail. Aside from the rare occurrence of lack of memory (ENOBUFS), a socket request may fail due to a request for an unknown protocol (EPROTONOSUPPORT), or a request for a type of socket for which there is no supporting protocol (EPROTOTYPE).

2.3. Binding local names

A socket is created without a name. Until a name is bound to a socket, processes have no way to reference it and, consequently, no messages may be received on it. Communicating processes are bound by an *association.* In the Internet and NS domains, an association is composed of local and foreign addresses, and local and foreign ports, while in the UNIX domain, an association is composed of local and foreign path names (the phrase "foreign pathname" means a pathname created by a foreign process, not a pathname on a foreign system). In most domains, associations must be unique. In the Internet domain there may never be duplicate <protocol, local address, local port, foreign address, foreign port> tuples. UNIX domain sockets need not always be bound to a name, but when bound there may never be duplicate <protocol, local pathname, foreign pathname> tuples. The pathnames may not refer to files already existing on the system in 4.3; the situation may change in future releases.

The *bind* system call allows a process to specify half of an association, <local address, local port> (or <local pathname>), while the *connect* and *accept* primitives are used to complete a socket's association.

* The manifest constants are named AF_whatever as they indicate the "address format" to use in interpreting names.

In the Internet domain, binding names to sockets can be fairly complex. Fortunately, it is usually not necessary to specifically bind an address and port number to a socket, because the *connect* and *send* calls will automatically bind an appropriate address if they are used with an unbound socket. The process of binding names to NS sockets is similar in most ways to that of binding names to Internet sockets.

The *bind* system call is used as follows:

```
bind(s, name, namelen);
```

The bound name is a variable length byte string which is interpreted by the supporting protocol(s). Its interpretation may vary from communication domain to communication domain (this is one of the properties which comprise the "domain"). As mentioned, in the Internet domain names contain an Internet address and port number. NS domain names contain an NS address and port number. In the UNIX domain, names contain a path name and a family, which is always AF_UNIX. If one wanted to bind the name "/tmp/foo" to a UNIX domain socket, the following code would be used*:

```
#include <sys/un.h>
...
struct sockaddr_un addr;
...
strcpy(addr.sun_path, "/tmp/foo");
addr.sun_family = AF_UNIX;
bind(s, (struct sockaddr *) &addr, strlen(addr.sun_path) +
    sizeof (addr.sun_family));
```

Note that in determining the size of a UNIX domain address null bytes are not counted, which is why *strlen* is used. In the current implementation of UNIX domain IPC under 4.3BSD, the file name referred to in *addr.sun_path* is created as a socket in the system file space. The caller must, therefore, have write permission in the directory where *addr.sun_path* is to reside, and this file should be deleted by the caller when it is no longer needed. Future versions of 4BSD may not create this file.

In binding an Internet address things become more complicated. The actual call is similar,

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct sockaddr_in sin;
...
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

but the selection of what to place in the address *sin* requires some discussion. We will come back to the problem of formulating Internet addresses in section 3 when the library routines used in name resolution are discussed.

Binding an NS address to a socket is even more difficult, especially since the Internet library routines do not work with NS hostnames. The actual call is again similar:

```
#include <sys/types.h>
#include <netns/ns.h>
...
struct sockaddr_ns sns;
...
bind(s, (struct sockaddr *) &sns, sizeof (sns));
```

Again, discussion of what to place in a "struct sockaddr_ns" will be deferred to section 3.

* Note that, although the tendency here is to call the "addr" structure "sun", doing so would cause problems if the code were ever ported to a Sun workstation.

2.4. Connection establishment

Connection establishment is usually asymmetric, with one process a "client" and the other a "server". The server, when willing to offer its advertised services, binds a socket to a well-known address associated with the service and then passively "listens" on its socket. It is then possible for an unrelated process to rendezvous with the server. The client requests services from the server by initiating a "connection" to the server's socket. On the client side the *connect* call is used to initiate a connection. Using the UNIX domain, this might appear as,

```
struct sockaddr_un server;
...
connect(s, (struct sockaddr *)&server, strlen(server.sun_path) +
    sizeof (server.sun_family));
```

while in the Internet domain,

```
struct sockaddr_in server;
...
connect(s, (struct sockaddr *)&server, sizeof (server));
```

and in the NS domain,

```
struct sockaddr_ns server;
...
connect(s, (struct sockaddr *)&server, sizeof (server));
```

where *server* in the example above would contain either the UNIX pathname, Internet address and port number, or NS address and port number of the server to which the client process wishes to speak. If the client process's socket is unbound at the time of the connect call, the system will automatically select and bind a name to the socket if necessary; c.f. section 5.4. This is the usual way that local addresses are bound to a socket.

An error is returned if the connection was unsuccessful (any name automatically bound by the system, however, remains). Otherwise, the socket is associated with the server and data transfer may begin. Some of the more common errors returned when a connection attempt fails are:

ETIMEDOUT
: After failing to establish a connection for a period of time, the system decided there was no point in retrying the connection attempt any more. This usually occurs because the destination host is down, or because problems in the network resulted in transmissions being lost.

ECONNREFUSED
: The host refused service for some reason. This is usually due to a server process not being present at the requested name.

ENETDOWN or EHOSTDOWN
: These operational errors are returned based on status information delivered to the client host by the underlying communication services.

ENETUNREACH or EHOSTUNREACH
: These operational errors can occur either because the network or host is unknown (no route to the network or host is present), or because of status information returned by intermediate gateways or switching nodes. Many times the status returned is not sufficient to distinguish a network being down from a host being down, in which case the system indicates the entire network is unreachable.

For the server to receive a client's connection it must perform two steps after binding its socket. The first is to indicate a willingness to listen for incoming connection requests:

```
listen(s, 5);
```

The second parameter to the *listen* call specifies the maximum number of outstanding connections which may be queued awaiting acceptance by the server process; this number may be limited by the

system. Should a connection be requested while the queue is full, the connection will not be refused, but rather the individual messages which comprise the request will be ignored. This gives a harried server time to make room in its pending connection queue while the client retries the connection request. Had the connection been returned with the ECONNREFUSED error, the client would be unable to tell if the server was up or not. As it is now it is still possible to get the ETIMEDOUT error back, though this is unlikely. The backlog figure supplied with the listen call is currently limited by the system to a maximum of 5 pending connections on any one queue. This avoids the problem of processes hogging system resources by setting an infinite backlog, then ignoring all connection requests.

With a socket marked as listening, a server may *accept* a connection:

```
struct sockaddr_in from;
...
fromlen = sizeof (from);
newsock = accept(s, (struct sockaddr *)&from, &fromlen);
```

(For the UNIX domain, *from* would be declared as a *struct sockaddr_un*, and for the NS domain, *from* would be declared as a *struct sockaddr_ns*, but nothing different would need to be done as far as *fromlen* is concerned. In the examples which follow, only Internet routines will be discussed.) A new descriptor is returned on receipt of a connection (along with a new socket). If the server wishes to find out who its client is, it may supply a buffer for the client socket's name. The value-result parameter *fromlen* is initialized by the server to indicate how much space is associated with *from*, then modified on return to reflect the true size of the name. If the client's name is not of interest, the second parameter may be a null pointer.

Accept normally blocks. That is, *accept* will not return until a connection is available or the system call is interrupted by a signal to the process. Further, there is no way for a process to indicate it will accept connections from only a specific individual, or individuals. It is up to the user process to consider who the connection is from and close down the connection if it does not wish to speak to the process. If the server process wants to accept connections on more than one socket, or wants to avoid blocking on the accept call, there are alternatives; they will be considered in section 5.

2.5. Data transfer

With a connection established, data may begin to flow. To send and receive data there are a number of possible calls. With the peer entity at each end of a connection anchored, a user can send or receive a message without specifying the peer. As one might expect, in this case, then the normal *read* and *write* system calls are usable,

```
write(s, buf, sizeof (buf));
read(s, buf, sizeof (buf));
```

In addition to *read* and *write*, the new calls *send* and *recv* may be used:

```
send(s, buf, sizeof (buf), flags);
recv(s, buf, sizeof (buf), flags);
```

While *send* and *recv* are virtually identical to *read* and *write*, the extra *flags* argument is important. The flags, defined in *<sys/socket.h>*, may be specified as a non-zero value if one or more of the following is required:

MSG_OOB	send/receive out of band data
MSG_PEEK	look at data without reading
MSG_DONTROUTE	send data without routing packets

Out of band data is a notion specific to stream sockets, and one which we will not immediately consider. The option to have data sent without routing applied to the outgoing packets is currently used only by the routing table management process, and is unlikely to be of interest to the casual user. The ability to preview data is, however, of interest. When MSG_PEEK is specified with a *recv* call,

any data present is returned to the user, but treated as still "unread". That is, the next *read* or *recv* call applied to the socket will return the data previously previewed.

2.6. Discarding sockets

Once a socket is no longer of interest, it may be discarded by applying a *close* to the descriptor,

```
close(s);
```

If data is associated with a socket which promises reliable delivery (e.g. a stream socket) when a close takes place, the system will continue to attempt to transfer the data. However, after a fairly long period of time, if the data is still undelivered, it will be discarded. Should a user have no use for any pending data, it may perform a *shutdown* on the socket prior to closing it. This call is of the form:

```
shutdown(s, how);
```

where *how* is 0 if the user is no longer interested in reading data, 1 if no more data will be sent, or 2 if no data is to be sent or received.

2.7. Connectionless sockets

To this point we have been concerned mostly with sockets which follow a connection oriented model. However, there is also support for connectionless interactions typical of the datagram facilities found in contemporary packet switched networks. A datagram socket provides a symmetric interface to data exchange. While processes are still likely to be client and server, there is no requirement for connection establishment. Instead, each message includes the destination address.

Datagram sockets are created as before. If a particular local address is needed, the *bind* operation must precede the first data transmission. Otherwise, the system will set the local address and/or port when data is first sent. To send data, the *sendto* primitive is used,

```
sendto(s, buf, buflen, flags, (struct sockaddr *)&to, tolen);
```

The *s*, *buf*, *buflen*, and *flags* parameters are used as before. The *to* and *tolen* values are used to indicate the address of the intended recipient of the message. When using an unreliable datagram interface, it is unlikely that any errors will be reported to the sender. When information is present locally to recognize a message that can not be delivered (for instance when a network is unreachable), the call will return −1 and the global value *errno* will contain an error number.

To receive messages on an unconnected datagram socket, the *recvfrom* primitive is provided:

```
recvfrom(s, buf, buflen, flags, (struct sockaddr *)&from, &fromlen);
```

Once again, the *fromlen* parameter is handled in a value-result fashion, initially containing the size of the *from* buffer, and modified on return to indicate the actual size of the address from which the datagram was received.

In addition to the two calls mentioned above, datagram sockets may also use the *connect* call to associate a socket with a specific destination address. In this case, any data sent on the socket will automatically be addressed to the connected peer, and only data received from that peer will be delivered to the user. Only one connected address is permitted for each socket at one time; a second connect will change the destination address, and a connect to a null address (family AF_UNSPEC) will disconnect. Connect requests on datagram sockets return immediately, as this simply results in the system recording the peer's address (as compared to a stream socket, where a connect request initiates establishment of an end to end connection). *Accept* and *listen* are not used with datagram sockets.

While a datagram socket socket is connected, errors from recent *send* calls may be returned asynchronously. These errors may be reported on subsequent operations on the socket, or a special socket option used with *getsockopt*, SO_ERROR, may be used to interrogate the error status. A *select* for reading or writing will return true when an error indication has been received. The next operation will return the error, and the error status is cleared. Other of the less important details of datagram sockets are described in section 5.

2.8. Input/Output multiplexing

One last facility often used in developing applications is the ability to multiplex i/o requests among multiple sockets and/or files. This is done using the *select* call:

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

fd_set readmask, writemask, exceptmask;
struct timeval timeout;
...
select(nfds, &readmask, &writemask, &exceptmask, &timeout);
```

Select takes as arguments pointers to three sets, one for the set of file descriptors for which the caller wishes to be able to read data on, one for those descriptors to which data is to be written, and one for which exceptional conditions are pending; out-of-band data is the only exceptional condition currently implemented by the socket If the user is not interested in certain conditions (i.e., read, write, or exceptions), the corresponding argument to the *select* should be a null pointer.

Each set is actually a structure containing an array of long integer bit masks; the size of the array is set by the definition FD_SETSIZE. The array is be long enough to hold one bit for each of FD_SETSIZE file descriptors.

The macros FD_SET(*fd, &mask*) and FD_CLR(*fd, &mask*) have been provided for adding and removing file descriptor *fd* in the set *mask*. The set should be zeroed before use, and the macro FD_ZERO(*&mask*) has been provided to clear the set *mask*. The parameter *nfds* in the *select* call specifies the range of file descriptors (i.e. one plus the value of the largest descriptor) to be examined in a set.

A timeout value may be specified if the selection is not to last more than a predetermined period of time. If the fields in *timeout* are set to 0, the selection takes the form of a *poll*, returning immediately. If the last parameter is a null pointer, the selection will block indefinitely*. *Select* normally returns the number of file descriptors selected; if the *select* call returns due to the timeout expiring, then the value 0 is returned. If the *select* terminates because of an error or interruption, a –1 is returned with the error number in *errno*, and with the file descriptor masks unchanged.

Assuming a successful return, the three sets will indicate which file descriptors are ready to be read from, written to, or have exceptional conditions pending. The status of a file descriptor in a select mask may be tested with the *FD_ISSET(fd, &mask)* macro, which returns a non-zero value if *fd* is a member of the set *mask*, and 0 if it is not.

To determine if there are connections waiting on a socket to be used with an *accept* call, *select* can be used, followed by a *FD_ISSET(fd, &mask)* macro to check for read readiness on the appropriate socket. If *FD_ISSET* returns a non-zero value, indicating permission to read, then a connection is pending on the socket.

As an example, to read data from two sockets, *s1* and *s2* as it is available from each and with a one-second timeout, the following code might be used:

* To be more specific, a return takes place only when a descriptor is selectable, or when a signal is received by the caller, interrupting the system call.

```
#include <sys/time.h>
#include <sys/types.h>
 ...
fd_set read_template;
struct timeval wait;
 ...
for (;;) {
        wait.tv_sec = 1;        /* one second */
        wait.tv_usec = 0;

        FD_ZERO(&read_template);

        FD_SET(s1, &read_template);
        FD_SET(s2, &read_template);

        nb = select(FD_SETSIZE, &read_template, (fd_set *) 0, (fd_set *) 0, &wait);
        if (nb <= 0) {
                An error occurred during the select, or
                the select timed out.
        }

        if (FD_ISSET(s1, &read_template)) {
                Socket #1 is ready to be read from.
        }

        if (FD_ISSET(s2, &read_template)) {
                Socket #2 is ready to be read from.
        }
}
```

In 4.2, the arguments to *select* were pointers to integers instead of pointers to *fd_sets*. This type of call will still work as long as the number of file descriptors being examined is less than the number of bits in an integer; however, the methods illustrated above should be used in all current programs.

Select provides a synchronous multiplexing scheme. Asynchronous notification of output completion, input availability, and exceptional conditions is possible through use of the SIGIO and SIGURG signals described in section 5.

3. NETWORK LIBRARY ROUTINES

The discussion in section 2 indicated the possible need to locate and construct network addresses when using the interprocess communication facilities in a distributed environment. To aid in this task a number of routines have been added to the standard C run-time library. In this section we will consider the new routines provided to manipulate network addresses. While the 4.3BSD networking facilities support both the DARPA standard Internet protocols and the Xerox NS protocols, most of the routines presented in this section do not apply to the NS domain. Unless otherwise stated, it should be assumed that the routines presented in this section do not apply to the NS domain.

Locating a service on a remote host requires many levels of mapping before client and server may communicate. A service is assigned a name which is intended for human consumption; e.g. "the *login server* on host monet". This name, and the name of the peer host, must then be translated into network *addresses* which are not necessarily suitable for human consumption. Finally, the address must then used in locating a physical *location* and *route* to the service. The specifics of these three mappings are likely to vary between network architectures. For instance, it is desirable for a network to not require hosts to be named in such a way that their physical location is known by the client host. Instead, underlying services in the network may discover the actual location of the host at the time a client host wishes to communicate. This ability to have hosts named in a location independent manner may induce overhead in connection establishment, as a discovery process must take place, but allows a host to be physically mobile without requiring it to notify its clientele of its current location.

Standard routines are provided for: mapping host names to network addresses, network names to network numbers, protocol names to protocol numbers, and service names to port numbers and the appropriate protocol to use in communicating with the server process. The file *<netdb.h>* must be included when using any of these routines.

3.1. Host names

An Internet host name to address mapping is represented by the *hostent* structure:

```
struct    hostent {
              char    *h_name;          /* official name of host */
              char    **h_aliases;      /* alias list */
              int     h_addrtype;       /* host address type (e.g., AF_INET) */
              int     h_length;         /* length of address */
              char    **h_addr_list;    /* list of addresses, null terminated */
};

#define   h_addr h_addr_list[0]         /* first address, network byte order */
```

The routine *gethostbyname*(3N) takes an Internet host name and returns a *hostent* structure, while the routine *gethostbyaddr*(3N) maps Internet host addresses into a *hostent* structure.

The official name of the host and its public aliases are returned by these routines, along with the address type (family) and a null terminated list of variable length address. This list of addresses is required because it is possible for a host to have many addresses, all having the same name. The *h_addr* definition is provided for backward compatibility, and is defined to be the first address in the list of addresses in the *hostent* structure.

The database for these calls is provided either by the file */etc/hosts* (*hosts*(5)), or by use of a nameserver, *named*(8). Because of the differences in these databases and their access protocols, the information returned may differ. When using the host table version of *gethostbyname*, only one address will be returned, but all listed aliases will be included. The nameserver version may return alternate addresses, but will not provide any aliases other than one given as argument.

Unlike Internet names, NS names are always mapped into host addresses by the use of a standard NS *Clearinghouse service*, a distributed name and authentication server. The algorithms for mapping NS names to addresses via a Clearinghouse are rather complicated, and the routines are not part of the standard libraries. The user-contributed Courier (Xerox remote procedure call protocol) compiler contains routines to accomplish this mapping; see the documentation and examples provided therein for more information. It is expected that almost all software that has to communicate using NS will need to use the facilities of the Courier compiler.

An NS host address is represented by the following:

```
union ns_host {
        u_char      c_host[6];
        u_short     s_host[3];
};

union ns_net {
        u_char      c_net[4];
        u_short     s_net[2];
};

struct ns_addr {
        union ns_net    x_net;
        union ns_host   x_host;
        u_short     x_port;
};
```

The following code fragment inserts a known NS address into a *ns_addr*:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
 ...
u_long netnum;
struct sockaddr_ns dst;
 ...
bzero((char *)&dst, sizeof(dst));

/*
 * There is no convenient way to assign a long
 * integer to a "union ns_net" at present; in
 * the future, something will hopefully be provided,
 * but this is the portable way to go for now.
 * The network number below is the one for the NS net
 * that the desired host (gyre) is on.
 */
netnum = htonl(2266);
dst.sns_addr.x_net = *(union ns_net *) &netnum;
dst.sns_family = AF_NS;

/*
 * host 2.7.1.0.2a.18 == "gyre:Computer Science:UofMaryland"
 */
dst.sns_addr.x_host.c_host[0] = 0x02;
dst.sns_addr.x_host.c_host[1] = 0x07;
dst.sns_addr.x_host.c_host[2] = 0x01;
dst.sns_addr.x_host.c_host[3] = 0x00;
dst.sns_addr.x_host.c_host[4] = 0x2a;
dst.sns_addr.x_host.c_host[5] = 0x18;
dst.sns_addr.x_port = htons(75);
```

3.2. Network names

As for host names, routines for mapping network names to numbers, and back, are provided. These routines return a *netent* structure:

```
/*
 * Assumption here is that a network number
 * fits in 32 bits -- probably a poor one.
 */
struct  netent {
        char        *n_name;        /* official name of net */
        char        **n_aliases;    /* alias list */
        int         n_addrtype;     /* net address type */
        int         n_net;          /* network number, host byte order */
};
```

The routines *getnetbyname*(3N), *getnetbynumber*(3N), and *getnetent*(3N) are the network counterparts to the host routines described above. The routines extract their information from */etc/networks.*

NS network numbers are determined either by asking your local Xerox Network Administrator (and hardcoding the information into your code), or by querying the Clearinghouse for addresses. The internetwork router is the only process that needs to manipulate network numbers on a regular basis; if a process wishes to communicate with a machine, it should ask the Clearinghouse for that machine's address (which will include the net number).

3.3. Protocol names

For protocols, which are defined in */etc/protocols*, the *protoent* structure defines the protocol-name mapping used with the routines *getprotobyname*(3N), *getprotobynumber*(3N), and *getprotoent*(3N):

```
struct  protoent {
        char    *p_name;        /* official protocol name */
        char    **p_aliases;    /* alias list */
        int     p_proto;        /* protocol number */
};
```

In the NS domain, protocols are indicated by the "client type" field of a IDP header. No protocol database exists; see section 5 for more information.

3.4. Service names

Information regarding services is a bit more complicated. A service is expected to reside at a specific "port" and employ a particular communication protocol. This view is consistent with the Internet domain, but inconsistent with other network architectures. Further, a service may reside on multiple ports. If this occurs, the higher level library routines will have to be bypassed or extended. Services available are contained in the file */etc/services*. A service mapping is described by the *servent* structure,

```
struct  servent {
        char    *s_name;        /* official service name */
        char    **s_aliases;    /* alias list */
        int     s_port;         /* port number, network byte order */
        char    *s_proto;       /* protocol to use */
};
```

The routine *getservbyname*(3N) maps service names to a servent structure by specifying a service name and, optionally, a qualifying protocol. Thus the call

```
sp = getservbyname("telnet", (char *) 0);
```

returns the service specification for a telnet server using any protocol, while the call

```
sp = getservbyname("telnet", "tcp");
```

returns only that telnet server which uses the TCP protocol. The routines *getservbyport*(3N) and *getservent*(3N) are also provided. The *getservbyport* routine has an interface similar to that provided by *getservbyname*; an optional protocol name may be specified to qualify lookups.

In the NS domain, services are handled by a central dispatcher provided as part of the Courier remote procedure call facilities. Again, the reader is referred to the Courier compiler documentation and to the Xerox standard* for further details.

3.5. Miscellaneous

With the support routines described above, an Internet application program should rarely have to deal directly with addresses. This allows services to be developed as much as possible in a network independent fashion. It is clear, however, that purging all network dependencies is very difficult. So long as the user is required to supply network addresses when naming services and sockets there will always some network dependency in a program. For example, the normal code included in client programs, such as the remote login program, is of the form shown in Figure 1. (This example will be considered in more detail in section 4.)

* *Courier: The Remote Procedure Call Protocol*, XSIS 038112.

If we wanted to make the remote login program independent of the Internet protocols and addressing scheme we would be forced to add a layer of routines which masked the network dependent aspects from the mainstream login code. For the current facilities available in the system this does not appear to be worthwhile.

Aside from the address-related data base routines, there are several other routines available in the run-time library which are of interest to users. These are intended mostly to simplify manipulation of names and addresses. Table 1 summarizes the routines for manipulating variable length byte strings and handling byte swapping of network addresses and values.

Call	Synopsis
bcmp(s1, s2, n)	compare byte-strings; 0 if same, not 0 otherwise
bcopy(s1, s2, n)	copy n bytes from s1 to s2
bzero(base, n)	zero-fill n bytes starting at base
htonl(val)	convert 32-bit quantity from host to network byte order
htons(val)	convert 16-bit quantity from host to network byte order
ntohl(val)	convert 32-bit quantity from network to host byte order
ntohs(val)	convert 16-bit quantity from network to host byte order

Table 1. C run-time routines.

The byte swapping routines are provided because the operating system expects addresses to be supplied in network order. On some architectures, such as the VAX, host byte ordering is different than network byte ordering. Consequently, programs are sometimes required to byte swap quantities. The library routines which return network addresses provide them in network order so that they may simply be copied into the structures provided to the system. This implies users should encounter the byte swapping problem only when *interpreting* network addresses. For example, if an Internet port is to be printed out the following code would be required:

```
printf("port number %d\n", ntohs(sp->s_port));
```

On machines where unneeded these routines are defined as null macros.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
 ...
main(argc, argv)
        int argc;
        char *argv[];
{
        struct sockaddr_in server;
        struct servent *sp;
        struct hostent *hp;
        int s;
        ...
        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogin: tcp/login: unknown service\n");
                exit(1);
        }
        hp = gethostbyname(argv[1]);
        if (hp == NULL) {
                fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
                exit(2);
        }
        bzero((char *)&server, sizeof (server));
        bcopy(hp->h_addr, (char *)&server.sin_addr, hp->h_length);
        server.sin_family = hp->h_addrtype;
        server.sin_port = sp->s_port;
        s = socket(AF_INET, SOCK_STREAM, 0);
        if (s < 0) {
                perror("rlogin: socket");
                exit(3);
        }
        ...
        /* Connect does the bind() for us */

        if (connect(s, (char *)&server, sizeof (server)) < 0) {
                perror("rlogin: connect");
                exit(5);
        }
        ...
}
```

Figure 1. Remote login client code.

4. CLIENT/SERVER MODEL

The most commonly used paradigm in constructing distributed applications is the client/server model. In this scheme client applications request services from a server process. This implies an asymmetry in establishing communication between the client and server which has been examined in section 2. In this section we will look more closely at the interactions between client and server, and consider some of the problems in developing client and server applications.

The client and server require a well known set of conventions before service may be rendered (and accepted). This set of conventions comprises a protocol which must be implemented at both ends of a connection. Depending on the situation, the protocol may be symmetric or asymmetric. In a symmetric protocol, either side may play the master or slave roles. In an asymmetric protocol, one side is immutably recognized as the master, with the other as the slave. An example of a symmetric protocol is the TELNET protocol used in the Internet for remote terminal emulation. An example of an asymmetric protocol is the Internet file transfer protocol, FTP. No matter whether the specific protocol used in obtaining a service is symmetric or asymmetric, when accessing a service there is a "client process" and a "server process". We will first consider the properties of server processes, then client processes.

A server process normally listens at a well known address for service requests. That is, the server process remains dormant until a connection is requested by a client's connection to the server's address. At such a time the server process "wakes up" and services the client, performing whatever appropriate actions the client requests of it.

Alternative schemes which use a service server may be used to eliminate a flock of server processes clogging the system while remaining dormant most of the time. For Internet servers in 4.3BSD, this scheme has been implemented via *inetd*, the so called "internet super-server." *Inetd* listens at a variety of ports, determined at start-up by reading a configuration file. When a connection is requested to a port on which *inetd* is listening, *inetd* executes the appropriate server program to handle the client. With this method, clients are unaware that an intermediary such as *inetd* has played any part in the connection. *Inetd* will be described in more detail in section 5.

A similar alternative scheme is used by most Xerox services. In general, the Courier dispatch process (if used) accepts connections from processes requesting services of some sort or another. The client processes request a particular <program number, version number, procedure number> triple. If the dispatcher knows of such a program, it is started to handle the request; if not, an error is reported to the client. In this way, only one port is required to service a large variety of different requests. Again, the Courier facilities are not available without the use and installation of the Courier compiler. The information presented in this section applies only to NS clients and services that do not use Courier.

4.1. Servers

In 4.3BSD most servers are accessed at well known Internet addresses or UNIX domain names. For example, the remote login server's main loop is of the form shown in Figure 2.

The first step taken by the server is look up its service definition:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogind: tcp/login: unknown service\n");
        exit(1);
}
```

The result of the *getservbyname* call is used in later portions of the code to define the Internet port at which it listens for service requests (indicated by a connection).

```
main(argc, argv)
        int argc;
        char *argv[];
{
        int f;
        struct sockaddr_in from;
        struct servent *sp;

        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogind: tcp/login: unknown service\n");
                exit(1);
        }
        ...
#ifndef DEBUG
        /* Disassociate server from controlling terminal */
        ...
#endif

        sin.sin_port = sp->s_port;      /* Restricted port -- see section 5 */
        ...
        f = socket(AF_INET, SOCK_STREAM, 0);
        ...
        if (bind(f, (struct sockaddr *) &sin, sizeof (sin)) < 0) {
                ...
        }
        ...
        listen(f, 5);
        for (;;) {
                int g, len = sizeof (from);

                g = accept(f, (struct sockaddr *) &from, &len);
                if (g < 0) {
                        if (errno != EINTR)
                                syslog(LOG_ERR, "rlogind: accept: %m");
                        continue;
                }
                if (fork() == 0) {
                        close(f);
                        doit(g, &from);
                }
                close(g);
        }
}
```

Figure 2. Remote login server.

Step two is to disassociate the server from the controlling terminal of its invoker:

```
for (i = 0; i < 3; ++i)
        close(i);

open("/", O_RDONLY);
dup2(0, 1);
dup2(0, 2);

i = open("/dev/tty", O_RDWR);
if (i >= 0) {
        ioctl(i, TIOCNOTTY, 0);
        close(i);
}
```

This step is important as the server will likely not want to receive signals delivered to the process group of the controlling terminal. Note, however, that once a server has disassociated itself it can no longer send reports of errors to a terminal, and must log errors via *syslog*.

Once a server has established a pristine environment, it creates a socket and begins accepting service requests. The *bind* call is required to insure the server listens at its expected location. It should be noted that the remote login server listens at a restricted port number, and must therefore be run with a user-id of root. This concept of a "restricted port number" is 4BSD specific, and is covered in section 5.

The main body of the loop is fairly simple:

```
for (;;) {
        int g, len = sizeof (from);

        g = accept(f, (struct sockaddr *)&from, &len);
        if (g < 0) {
                if (errno != EINTR)
                        syslog(LOG_ERR, "rlogind: accept: %m");
                continue;
        }
        if (fork() == 0) {          /* Child */
                close(f);
                doit(g, &from);
        }
        close(g);                   /* Parent */
}
```

An *accept* call blocks the server until a client requests service. This call could return a failure status if the call is interrupted by a signal such as SIGCHLD (to be discussed in section 5). Therefore, the return value from *accept* is checked to insure a connection has actually been established, and an error report is logged via *syslog* if an error has occurred.

With a connection in hand, the server then forks a child process and invokes the main body of the remote login protocol processing. Note how the socket used by the parent for queuing connection requests is closed in the child, while the socket created as a result of the *accept* is closed in the parent. The address of the client is also handed the *doit* routine because it requires it in authenticating clients.

4.2. Clients

The client side of the remote login service was shown earlier in Figure 1. One can see the separate, asymmetric roles of the client and server clearly in the code. The server is a passive entity, listening for client connections, while the client process is an active entity, initiating a connection

when invoked.

Let us consider more closely the steps taken by the client remote login process. As in the server process, the first step is to locate the service definition for a remote login:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogin: tcp/login: unknown service\n");
        exit(1);
}
```

Next the destination host is looked up with a *gethostbyname* call:

```
hp = gethostbyname(argv[1]);
if (hp == NULL) {
        fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
        exit(2);
}
```

With this accomplished, all that is required is to establish a connection to the server at the requested host and start up the remote login protocol. The address buffer is cleared, then filled in with the Internet address of the foreign host and the port number at which the login process resides on the foreign host:

```
bzero((char *)&server, sizeof (server));
bcopy(hp->h_addr, (char *) &server.sin_addr, hp->h_length);
server.sin_family = hp->h_addrtype;
server.sin_port = sp->s_port;
```

A socket is created, and a connection initiated. Note that *connect* implicitly performs a *bind* call, since *s* is unbound.

```
s = socket(hp->h_addrtype, SOCK_STREAM, 0);
if (s < 0) {
        perror("rlogin: socket");
        exit(3);
}
...
if (connect(s, (struct sockaddr *) &server, sizeof (server)) < 0) {
        perror("rlogin: connect");
        exit(4);
}
```

The details of the remote login protocol will not be considered here.

4.3. Connectionless servers

While connection-based services are the norm, some services are based on the use of datagram sockets. One, in particular, is the "rwho" service which provides users with status information for hosts connected to a local area network. This service, while predicated on the ability to *broadcast* information to all hosts connected to a particular network, is of interest as an example usage of datagram sockets.

A user on any machine running the rwho server may find out the current status of a machine with the *ruptime*(1) program. The output generated is illustrated in Figure 3.

Status information for each host is periodically broadcast by rwho server processes on each machine. The same server process also receives the status information and uses it to update a database. This database is then interpreted to generate the status information for each host. Servers operate autonomously, coupled only by the local network and its broadcast capabilities.

```
arpa      up   9:45,     5 users, load   1.15,  1.39,  1.31
cad       up   2+12:04,  8 users, load   4.67,  5.13,  4.59
calder    up   10:10,    0 users, load   0.27,  0.15,  0.14
dali      up   2+06:28,  9 users, load   1.04,  1.20,  1.65
degas     up   25+09:48, 0 users, load   1.49,  1.43,  1.41
ear       up   5+00:05,  0 users, load   1.51,  1.54,  1.56
ernie     down 0:24
esvax     down 17:04
ingres    down 0:26
kim       up   3+09:16,  8 users, load   2.03,  2.46,  3.11
matisse   up   3+06:18,  0 users, load   0.03,  0.03,  0.05
medea     up   3+09:39,  2 users, load   0.35,  0.37,  0.50
merlin    down 19+15:37
miro      up   1+07:20,  7 users, load   4.59,  3.28,  2.12
monet     up   1+00:43,  2 users, load   0.22,  0.09,  0.07
oz        down 16:09
statvax   up   2+15:57,  3 users, load   1.52,  1.81,  1.86
ucbvax    up   9:34,     2 users, load   6.08,  5.16,  3.28
```

Figure 3. ruptime output.

Note that the use of broadcast for such a task is fairly inefficient, as all hosts must process each message, whether or not using an rwho server. Unless such a service is sufficiently universal and is frequently used, the expense of periodic broadcasts outweighs the simplicity.

The rwho server, in a simplified form, is pictured in Figure 4. There are two separate tasks performed by the server. The first task is to act as a receiver of status information broadcast by other hosts on the network. This job is carried out in the main loop of the program. Packets received at the rwho port are interrogated to insure they've been sent by another rwho server process, then are time stamped with their arrival time and used to update a file indicating the status of the host. When a host has not been heard from for an extended period of time, the database interpretation routines assume the host is down and indicate such on the status reports. This algorithm is prone to error as a server may be down while a host is actually up, but serves our current needs.

The second task performed by the server is to supply information regarding the status of its host. This involves periodically acquiring system status information, packaging it up in a message and broadcasting it on the local network for other rwho servers to hear. The supply function is triggered by a timer and runs off a signal. Locating the system status information is somewhat involved, but uninteresting. Deciding where to transmit the resultant packet is somewhat problematical, however.

Status information must be broadcast on the local network. For networks which do not support the notion of broadcast another scheme must be used to simulate or replace broadcasting. One possibility is to enumerate the known neighbors (based on the status messages received from other rwho servers). This, unfortunately, requires some bootstrapping information, for a server will have no idea what machines are its neighbors until it receives status messages from them. Therefore, if all machines on a net are freshly booted, no machine will have any known neighbors and thus never receive, or send, any status information. This is the identical problem faced by the routing table management process in propagating routing status information. The standard solution, unsatisfactory as it may be, is to inform one or more servers of known neighbors and request that they always communicate with these neighbors. If each server has at least one neighbor supplied to it, status information may then propagate through a neighbor to hosts which are not (possibly) directly neighbors. If the server is able to support networks which provide a broadcast capability, as well as those which do not, then networks with an arbitrary topology may share status information*.

* One must, however, be concerned about "loops". That is, if a host is connected to multiple networks, it

It is important that software operating in a distributed environment not have any site-dependent information compiled into it. This would require a separate copy of the server at each host and make maintenance a severe headache. 4.3BSD attempts to isolate host-specific information from applications by providing system calls which return the necessary information*. A mechanism exists, in the form of an *ioctl* call, for finding the collection of networks to which a host is directly connected. Further, a local network broadcasting mechanism has been implemented at the socket level. Combining these two features allows a process to broadcast on any directly connected local network which supports the notion of broadcasting in a site independent manner. This allows 4.3BSD to solve the problem of deciding how to propagate status information in the case of *rwho*, or more generally in broadcasting: Such status information is broadcast to connected networks at the socket level, where the connected networks have been obtained via the appropriate *ioctl* calls. The specifics of such broadcastings are complex, however, and will be covered in section 5.

will receive status information from itself. This can lead to an endless, wasteful, exchange of information.
* An example of such a system call is the *gethostname*(2) call which returns the host's "official" name.

```
main()
{
        ...
        sp = getservbyname("who", "udp");
        net = getnetbyname("localnet");
        sin.sin_addr = inet_makeaddr(INADDR_ANY, net);
        sin.sin_port = sp->s_port;
        ...
        s = socket(AF_INET, SOCK_DGRAM, 0);
        ...
        on = 1;
        if (setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on)) < 0) {
                syslog(LOG_ERR, "setsockopt SO_BROADCAST: %m");
                exit(1);
        }
        bind(s, (struct sockaddr *) &sin, sizeof (sin));
        ...
        signal(SIGALRM, onalrm);
        onalrm();
        for (;;) {
                struct whod wd;
                int cc, whod, len = sizeof (from);

                cc = recvfrom(s, (char *)&wd, sizeof (struct whod), 0,
                   (struct sockaddr *)&from, &len);
                if (cc <= 0) {
                        if (cc < 0 && errno != EINTR)
                                syslog(LOG_ERR, "rwhod: recv: %m");
                        continue;
                }
                if (from.sin_port != sp->s_port) {
                        syslog(LOG_ERR, "rwhod: %d: bad from port",
                                ntohs(from.sin_port));
                        continue;
                }
                ...
                if (!verify(wd.wd_hostname)) {
                        syslog(LOG_ERR, "rwhod: malformed host name from %x",
                                ntohl(from.sin_addr.s_addr));
                        continue;
                }
                (void) sprintf(path, "%s/whod.%s", RWHODIR, wd.wd_hostname);
                whod = open(path, O_WRONLY | O_CREAT | O_TRUNC, 0666);
                ...
                (void) time(&wd.wd_recvtime);
                (void) write(whod, (char *)&wd, cc);
                (void) close(whod);
        }
}
```

Figure 4. rwho server.

5. ADVANCED TOPICS

A number of facilities have yet to be discussed. For most users of the IPC the mechanisms already described will suffice in constructing distributed applications. However, others will find the need to utilize some of the features which we consider in this section.

5.1. Out of band data

The stream socket abstraction includes the notion of "out of band" data. Out of band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out of band data is delivered to the user independently of normal data. The abstraction defines that the out of band data facilities must support the reliable delivery of at least one out of band message at a time. This message may contain at least one byte of data, and at least one message may be pending delivery to the user at any one time. For communications protocols which support only in-band signaling (i.e. the urgent data is delivered in sequence with the normal data), the system normally extracts the data from the normal data stream and stores it separately. This allows users to choose between receiving the urgent data in order and receiving it out of sequence without having to buffer all the intervening data. It is possible to "peek" (via MSG_PEEK) at out of band data. If the socket has a process group, a SIGURG signal is generated when the protocol is notified of its existence. A process can set the process group or process id to be informed by the SIGURG signal via the appropriate *fcntl* call, as described below for SIGIO. If multiple sockets may have out of band data awaiting delivery, a *select* call for exceptional conditions may be used to determine those sockets with such data pending. Neither the signal nor the select indicate the actual arrival of the out-of-band data, but only notification that it is pending.

In addition to the information passed, a logical mark is placed in the data stream to indicate the point at which the out of band data was sent. The remote login and remote shell applications use this facility to propagate signals between client and server processes. When a signal flushs any pending output from the remote process(es), all data up to the mark in the data stream is discarded.

To send an out of band message the MSG_OOB flag is supplied to a *send* or *sendto* calls, while to receive out of band data MSG_OOB should be indicated when performing a *recvfrom* or *recv* call. To find out if the read pointer is currently pointing at the mark in the data stream, the SIOCATMARK ioctl is provided:

```
ioctl(s, SIOCATMARK, &yes);
```

If *yes* is a 1 on return, the next read will return data after the mark. Otherwise (assuming out of band data has arrived), the next read will provide data sent by the client prior to transmission of the out of band signal. The routine used in the remote login process to flush output on receipt of an interrupt or quit signal is shown in Figure 5. It reads the normal data up to the mark (to discard it), then reads the out-of-band byte.

A process may also read or peek at the out-of-band data without first reading up to the mark. This is more difficult when the underlying protocol delivers the urgent data in-band with the normal data, and only sends notification of its presence ahead of time (e.g., the TCP protocol used to implement streams in the Internet domain). With such protocols, the out-of-band byte may not yet have arrived when a *recv* is done with the MSG_OOB flag. In that case, the call will return an error of EWOULDBLOCK. Worse, there may be enough in-band data in the input buffer that normal flow control prevents the peer from sending the urgent data until the buffer is cleared. The process must then read enough of the queued data that the urgent data may be delivered.

Certain programs that use multiple bytes of urgent data and must handle multiple urgent signals (e.g., *telnet*(1C)) need to retain the position of urgent data within the stream. This treatment is available as a socket-level option, SO_OOBINLINE; see *setsockopt*(2) for usage. With this option, the position of urgent data (the "mark") is retained, but the urgent data immediately follows the mark within the normal data stream returned without the MSG_OOB flag. Reception of multiple urgent indications causes the mark to move, but no out-of-band data are lost.

```
#include <sys/ioctl.h>
#include <sys/file.h>
...
oob()
{
        int out = FWRITE;
        char waste[BUFSIZ], mark;

        /* flush local terminal output */
        ioctl(1, TIOCFLUSH, (char *)&out);
        for (;;) {
                if (ioctl(rem, SIOCATMARK, &mark) < 0) {
                        perror("ioctl");
                        break;
                }
                if (mark)
                        break;
                (void) read(rem, waste, sizeof (waste));
        }
        if (recv(rem, &mark, 1, MSG_OOB) < 0) {
                perror("recv");
                ...
        }
        ...
}
```

Figure 5. Flushing terminal I/O on receipt of out of band data.

5.2. Non-Blocking Sockets

It is occasionally convenient to make use of sockets which do not block; that is, I/O requests which cannot complete immediately and would therefore cause the process to be suspended awaiting completion are not executed, and an error code is returned. Once a socket has been created via the *socket* call, it may be marked as non-blocking by *fcntl* as follows:

```
#include <fcntl.h>
...
int   s;
...
s = socket(AF_INET, SOCK_STREAM, 0);
...
if (fcntl(s, F_SETFL, FNDELAY) < 0)
        perror("fcntl F_SETFL, FNDELAY");
        exit(1);
}
...
```

When performing non-blocking I/O on sockets, one must be careful to check for the error EWOULDBLOCK (stored in the global variable *errno*), which occurs when an operation would normally block, but the socket it was performed on is marked as non-blocking. In particular, *accept*, *connect*, *send*, *recv*, *read*, and *write* can all return EWOULDBLOCK, and processes should be prepared to deal with such return codes. If an operation such as a *send* cannot be done in its entirety, but partial writes are sensible (for example, when using a stream socket), the data that can be sent immediately will be processed, and the return value will indicate the amount actually sent.

5.3. Interrupt driven socket I/O

The SIGIO signal allows a process to be notified via a signal when a socket (or more generally, a file descriptor) has data waiting to be read. Use of the SIGIO facility requires three steps: First, the process must set up a SIGIO signal handler by use of the *signal* or *sigvec* calls. Second, it must set the process id or process group id which is to receive notification of pending input to its own process id, or the process group id of its process group (note that the default process group of a socket is group zero). This is accomplished by use of an *fcntl* call. Third, it must enable asynchronous notification of pending I/O requests with another *fcntl* call. Sample code to allow a given process to receive information on pending I/O requests as they occur for a socket *s* is given in Figure 6. With the addition of a handler for SIGURG, this code can also be used to prepare for receipt of SIGURG signals.

```
#include <fcntl.h>
...
int   io_handler();
...
signal(SIGIO, io_handler);

/* Set the process receiving SIGIO/SIGURG signals to us */

if (fcntl(s, F_SETOWN, getpid()) < 0) {
      perror("fcntl F_SETOWN");
      exit(1);
}

/* Allow receipt of asynchronous I/O signals */

if (fcntl(s, F_SETFL, FASYNC) < 0) {
      perror("fcntl F_SETFL, FASYNC");
      exit(1);
}
```

Figure 6. Use of asynchronous notification of I/O requests.

5.4. Signals and process groups

Due to the existence of the SIGURG and SIGIO signals each socket has an associated process number, just as is done for terminals. This value is initialized to zero, but may be redefined at a later time with the F_SETOWN *fcntl*, such as was done in the code above for SIGIO. To set the socket's process id for signals, positive arguments should be given to the *fcntl* call. To set the socket's process group for signals, negative arguments should be passed to *fcntl*. Note that the process number indicates either the associated process id or the associated process group; it is impossible to specify both at the same time. A similar *fcntl*, F_GETOWN, is available for determining the current process number of a socket.

Another signal which is useful when constructing server processes is SIGCHLD. This signal is delivered to a process when any child processes have changed state. Normally servers use the signal to "reap" child processes that have exited without explicitly awaiting their termination or periodic polling for exit status. For example, the remote login server loop shown in Figure 2 may be augmented as shown in Figure 7.

If the parent server process fails to reap its children, a large number of "zombie" processes may be created.

```
int reaper();
 ...
signal(SIGCHLD, reaper);
listen(f, 5);
for (;;) {
        int g, len = sizeof (from);

        g = accept(f, (struct sockaddr *)&from, &len,);
        if (g < 0) {
                if (errno != EINTR)
                        syslog(LOG_ERR, "rlogind: accept: %m");
                continue;
        }
        ...
}
 ...
#include <wait.h>
reaper()
{
        union wait status;

        while (wait3(&status, WNOHANG, 0) > 0)
                ;
}
```

Figure 7. Use of the SIGCHLD signal.

5.5. Pseudo terminals

Many programs will not function properly without a terminal for standard input and output. Since sockets do not provide the semantics of terminals, it is often necessary to have a process communicating over the network do so through a *pseudo-terminal.* A pseudo- terminal is actually a pair of devices, master and slave, which allow a process to serve as an active agent in communication between processes and users. Data written on the slave side of a pseudo-terminal is supplied as input to a process reading from the master side, while data written on the master side are processed as terminal input for the slave. In this way, the process manipulating the master side of the pseudo-terminal has control over the information read and written on the slave side as if it were manipulating the keyboard and reading the screen on a real terminal. The purpose of this abstraction is to preserve terminal semantics over a network connection— that is, the slave side appears as a normal terminal to any process reading from or writing to it.

For example, the remote login server uses pseudo-terminals for remote login sessions. A user logging in to a machine across the network is provided a shell with a slave pseudo-terminal as standard input, output, and error. The server process then handles the communication between the programs invoked by the remote shell and the user's local client process. When a user sends a character that generates an interrupt on the remote machine that flushes terminal output, the pseudo-terminal generates a control message for the server process. The server then sends an out of band message to the client process to signal a flush of data at the real terminal and on the intervening data buffered in the network.

Under 4.3BSD, the name of the slave side of a pseudo-terminal is of the form */dev/ttyxy*, where *x* is a single letter starting at 'p' and continuing to 't'. *y* is a hexadecimal digit (i.e., a single character in the range 0 through 9 or 'a' through 'f'). The master side of a pseudo-terminal is */dev/ptyxy*, where *x* and *y* correspond to the slave side of the pseudo-terminal.

In general, the method of obtaining a pair of master and slave pseudo-terminals is to find a pseudo-terminal which is not currently in use. The master half of a pseudo-terminal is a single-open device; thus, each master may be opened in turn until an open succeeds. The slave side of the pseudo-terminal is then opened, and is set to the proper terminal modes if necessary. The process then *forks*; the child closes the master side of the pseudo-terminal, and *execs* the appropriate program. Meanwhile, the parent closes the slave side of the pseudo-terminal and begins reading and writing from the master side. Sample code making use of pseudo-terminals is given in Figure 8; this code assumes that a connection on a socket *s* exists, connected to a peer who wants a service of some kind, and that the process has disassociated itself from any previous controlling terminal.

5.6. Selecting specific protocols

If the third argument to the *socket* call is 0, *socket* will select a default protocol to use with the returned socket of the type requested. The default protocol is usually correct, and alternate choices are not usually available. However, when using "raw" sockets to communicate directly with lower-level protocols or hardware interfaces, the protocol argument may be important for setting up demultiplexing. For example, raw sockets in the Internet family may be used to implement a new protocol above IP, and the socket will receive packets only for the protocol specified. To obtain a particular protocol one determines the protocol number as defined within the communication domain. For the Internet domain one may use one of the library routines discussed in section 3, such as *getprotobyname*:

```
gotpty = 0;
for (c = 'p'; !gotpty && c <= 's'; c++) {
        line = "/dev/ptyXX";
        line[sizeof("/dev/pty")-1] = c;
        line[sizeof("/dev/ptyp")-1] = '0';
        if (stat(line, &statbuf) < 0)
                break;
        for (i = 0; i < 16; i++) {
                line[sizeof("/dev/ptyp")-1] = "0123456789abcdef"[i];
                master = open(line, O_RDWR);
                if (master > 0) {
                        gotpty = 1;
                        break;
                }
        }
}
if (!gotpty) {
        syslog(LOG_ERR, "All network ports in use");
        exit(1);
}

line[sizeof("/dev/")-1] = 't';
slave = open(line, O_RDWR);   /* slave is now slave side */
if (slave < 0) {
        syslog(LOG_ERR, "Cannot open slave pty %s", line);
        exit(1);
}

ioctl(slave, TIOCGETP, &b);     /* Set slave tty modes */
b.sg_flags = CRMOD|XTABS|ANYP;
ioctl(slave, TIOCSETP, &b);

i = fork();
if (i < 0) {
        syslog(LOG_ERR, "fork: %m");
        exit(1);
} else if (i) {             /* Parent */
        close(slave);
        ...
} else {              /* Child */
        (void) close(s);
        (void) close(master);
        dup2(slave, 0);
        dup2(slave, 1);
        dup2(slave, 2);
        if (slave > 2)
                (void) close(slave);
        ...
}
```

Figure 8. Creation and use of a pseudo terminal

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
...
pp = getprotobyname("newtcp");
s = socket(AF_INET, SOCK_STREAM, pp->p_proto);
```

This would result in a socket *s* using a stream based connection, but with protocol type of "newtcp" instead of the default "tcp."

In the NS domain, the available socket protocols are defined in *<netns/ns.h>*. To create a raw socket for Xerox Error Protocol messages, one might use:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
...
s = socket(AF_NS, SOCK_RAW, NSPROTO_ERROR);
```

5.7. Address binding

As was mentioned in section 2, binding addresses to sockets in the Internet and NS domains can be fairly complex. As a brief reminder, these associations are composed of local and foreign addresses, and local and foreign ports. Port numbers are allocated out of separate spaces, one for each system and one for each domain on that system. Through the *bind* system call, a process may specify half of an association, the <local address, local port> part, while the *connect* and *accept* primitives are used to complete a socket's association by specifying the <foreign address, foreign port> part. Since the association is created in two steps the association uniqueness requirement indicated previously could be violated unless care is taken. Further, it is unrealistic to expect user programs to always know proper values to use for the local address and local port since a host may reside on multiple networks and the set of allocated port numbers is not directly accessible to a user.

To simplify local address binding in the Internet domain the notion of a "wildcard" address has been provided. When an address is specified as INADDR_ANY (a manifest constant defined in <netinet/in.h>), the system interprets the address as "any valid address". For example, to bind a specific port number to a socket, but leave the local address unspecified, the following code might be used:

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct sockaddr_in sin;
...
s = socket(AF_INET, SOCK_STREAM, 0);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

Sockets with wildcarded local addresses may receive messages directed to the specified port number, and sent to any of the possible addresses assigned to a host. For example, if a host has addresses 128.32.0.4 and 10.0.0.78, and a socket is bound as above, the process will be able to accept connection requests which are addressed to 128.32.0.4 or 10.0.0.78. If a server process wished to only allow hosts on a given network connect to it, it would bind the address of the host on the appropriate network.

In a similar fashion, a local port may be left unspecified (specified as zero), in which case the system will select an appropriate port number for it. This shortcut will work both in the Internet and NS domains. For example, to bind a specific local address to a socket, but to leave the local port number unspecified:

```
hp = gethostbyname(hostname);
if (hp == NULL) {
    ...
}
bcopy(hp->h_addr, (char *) sin.sin_addr, hp->h_length);
sin.sin_port = htons(0);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

The system selects the local port number based on two criteria. The first is that on 4BSD systems, Internet ports below IPPORT_RESERVED (1024) (for the Xerox domain, 0 through 3000) are reserved for privileged users (i.e., the super user); Internet ports above IPPORT_USERRESERVED (50000) are reserved for non-privileged servers. The second is that the port number is not currently bound to some other socket. In order to find a free Internet port number in the privileged range the *rresvport* library routine may be used as follows to return a stream socket in with a privileged port number:

```
int lport = IPPORT_RESERVED - 1;
int s;
s = rresvport(&lport);
if (s < 0) {
    if (errno == EAGAIN)
        fprintf(stderr, "socket: all ports in use\n");
    else
        perror("rresvport: socket");
    ...
}
```

The restriction on allocating ports was done to allow processes executing in a "secure" environment to perform authentication based on the originating address and port number. For example, the *rlogin*(1) command allows users to log in across a network without being asked for a password, if two conditions hold: First, the name of the system the user is logging in from is in the file */etc/hosts.equiv* on the system he is logging in to (or the system name and the user name are in the user's *.rhosts* file in the user's home directory), and second, that the user's rlogin process is coming from a privileged port on the machine from which he is logging. The port number and network address of the machine from which the user is logging in can be determined either by the *from* result of the *accept* call, or from the *getpeername* call.

In certain cases the algorithm used by the system in selecting port numbers is unsuitable for an application. This is because associations are created in a two step process. For example, the Internet file transfer protocol, FTP, specifies that data connections must always originate from the same local port. However, duplicate associations are avoided by connecting to different foreign ports. In this situation the system would disallow binding the same local address and port number to a socket if a previous data connection's socket still existed. To override the default port selection algorithm, an option call must be performed prior to address binding:

```
...
int   on = 1;
...
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

With the above call, local addresses may be bound which are already in use. This does not violate the uniqueness requirement as the system still checks at connect time to be sure any other sockets

with the same local address and port do not have the same foreign address and port. If the association already exists, the error EADDRINUSE is returned.

5.8. Broadcasting and determining network configuration

By using a datagram socket, it is possible to send broadcast packets on many networks supported by the system. The network itself must support broadcast; the system provides no simulation of broadcast in software. Broadcast messages can place a high load on a network since they force every host on the network to service them. Consequently, the ability to send broadcast packets has been limited to sockets which are explicitly marked as allowing broadcasting. Broadcast is typically used for one of two reasons: it is desired to find a resource on a local network without prior knowledge of its address, or important functions such as routing require that information be sent to all accessible neighbors.

To send a broadcast message, a datagram socket should be created:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

or

```
s = socket(AF_NS, SOCK_DGRAM, 0);
```

The socket is marked as allowing broadcasting,

```
int  on = 1;

setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof (on));
```

and at least a port number should be bound to the socket:

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

or, for the NS domain,

```
sns.sns_family = AF_NS;
netnum = htonl(net);
sns.sns_addr.x_net = *(union ns_net *) &netnum; /* insert net number */
sns.sns_addr.x_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sns, sizeof (sns));
```

The destination address of the message to be broadcast depends on the network(s) on which the message is to be broadcast. The Internet domain supports a shorthand notation for broadcast on the local network, the address INADDR_BROADCAST (defined in *<netinet/in.h>*. To determine the list of addresses for all reachable neighbors requires knowledge of the networks to which the host is connected. Since this information should be obtained in a host-independent fashion and may be impossible to derive, 4.3BSD provides a method of retrieving this information from the system data structures. The SIOCGIFCONF *ioctl* call returns the interface configuration of a host in the form of a single *ifconf* structure; this structure contains a "data area" which is made up of an array of of *ifreq* structures, one for each network interface to which the host is connected. These structures are defined in *<net/if.h>* as follows:

```
struct ifconf {
        int     ifc_len;                        /* size of associated buffer */
        union {
                caddr_t ifcu_buf;
                struct  ifreq *ifcu_req;
        } ifc_ifcu;
};

#define ifc_buf ifc_ifcu.ifcu_buf               /* buffer address */
#define ifc_req ifc_ifcu.ifcu_req               /* array of structures returned */

#define IFNAMSIZ    16

struct ifreq {
        char    ifr_name[IFNAMSIZ];             /* if name, e.g. "en0" */
        union {
                struct  sockaddr ifru_addr;
                struct  sockaddr ifru_dstaddr;
                struct  sockaddr ifru_broadaddr;
                short   ifru_flags;
                caddr_t ifru_data;
        } ifr_ifru;
};

#define  ifr_addr       ifr_ifru.ifru_addr      /* address */
#define  ifr_dstaddr    ifr_ifru.ifru_dstaddr   /* other end of p-to-p link */
#define  ifr_broadaddr  ifr_ifru.ifru_broadaddr /* broadcast address */
#define  ifr_flags      ifr_ifru.ifru_flags     /* flags */
#define  ifr_data       ifr_ifru.ifru_data      /* for use by interface */
```

The actual call which obtains the interface configuration is

```
struct ifconf ifc;
char buf[BUFSIZ];

ifc.ifc_len = sizeof (buf);
ifc.ifc_buf = buf;
if (ioctl(s, SIOCGIFCONF, (char *) &ifc) < 0) {
    ...
}
```

After this call *buf* will contain one *ifreq* structure for each network to which the host is connected, and *ifc.ifc_len* will have been modified to reflect the number of bytes used by the *ifreq* structures.

For each structure there exists a set of "interface flags" which tell whether the network corresponding to that interface is up or down, point to point or broadcast, etc. The SIOCGIFFLAGS *ioctl* retrieves these flags for an interface specified by an *ifreq* structure as follows:

```
struct ifreq *ifr;

ifr = ifc.ifc_req;

for (n = ifc.ifc_len / sizeof (struct ifreq); --n >= 0; ifr++) {
    /*
     * We must be careful that we don't use an interface
     * devoted to an address family other than those intended;
     * if we were interested in NS interfaces, the
     * AF_INET would be AF_NS.
     */
    if (ifr->ifr_addr.sa_family != AF_INET)
        continue;
    if (ioctl(s, SIOCGIFFLAGS, (char *) ifr) < 0) {
        ...
    }
    /*
     * Skip boring cases.
     */
    if ((ifr->ifr_flags & IFF_UP) == 0 ||
        (ifr->ifr_flags & IFF_LOOPBACK) ||
        (ifr->ifr_flags & (IFF_BROADCAST | IFF_POINTTOPOINT)) == 0)
        continue;
```

Once the flags have been obtained, the broadcast address must be obtained. In the case of broadcast networks this is done via the SIOCGIFBRDADDR *ioctl*, while for point-to-point networks the address of the destination host is obtained with SIOCGIFDSTADDR.

```
struct sockaddr dst;

if (ifr->ifr_flags & IFF_POINTTOPOINT) {
    if (ioctl(s, SIOCGIFDSTADDR, (char *) ifr) < 0) {
        ...
    }
    bcopy((char *) ifr->ifr_dstaddr, (char *) &dst, sizeof (ifr->ifr_dstaddr));
} else if (ifr->ifr_flags & IFF_BROADCAST) {
    if (ioctl(s, SIOCGIFBRDADDR, (char *) ifr) < 0) {
        ...
    }
    bcopy((char *) ifr->ifr_broadaddr, (char *) &dst, sizeof (ifr->ifr_broadaddr));
}
```

After the appropriate *ioctl*'s have obtained the broadcast or destination address (now in *dst*), the *sendto* call may be used:

```
    sendto(s, buf, buflen, 0, (struct sockaddr *)&dst, sizeof (dst));
}
```

In the above loop one *sendto* occurs for every interface to which the host is connected that supports the notion of broadcast or point-to-point addressing. If a process only wished to send broadcast messages on a given network, code similar to that outlined above would be used, but the loop would need to find the correct destination address.

Received broadcast messages contain the senders address and port, as datagram sockets are bound before a message is allowed to go out.

5.9. Socket Options

It is possible to set and get a number of options on sockets via the *setsockopt* and *getsockopt* system calls. These options include such things as marking a socket for broadcasting, not to route, to linger on close, etc. The general forms of the calls are:

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

and

```
getsockopt(s, level, optname, optval, optlen);
```

The parameters to the calls are as follows: *s* is the socket on which the option is to be applied. *Level* specifies the protocol layer on which the option is to be applied; in most cases this is the "socket level", indicated by the symbolic constant SOL_SOCKET, defined in *<sys/socket.h>*. The actual option is specified in *optname*, and is a symbolic constant also defined in *<sys/socket.h>*. *Optval* and *Optlen* point to the value of the option (in most cases, whether the option is to be turned on or off), and the length of the value of the option, respectively. For *getsockopt*, *optlen* is a value-result parameter, initially set to the size of the storage area pointed to by *optval*, and modified upon return to indicate the actual amount of storage used.

An example should help clarify things. It is sometimes useful to determine the type (e.g., stream, datagram, etc.) of an existing socket; programs under *inetd* (described below) may need to perform this task. This can be accomplished as follows via the SO_TYPE socket option and the *getsockopt* call:

```
#include <sys/types.h>
#include <sys/socket.h>

int type, size;

size = sizeof (int);

if (getsockopt(s, SOL_SOCKET, SO_TYPE, (char *) &type, &size) < 0) {
        ...
}
```

After the *getsockopt* call, *type* will be set to the value of the socket type, as defined in *<sys/socket.h>*. If, for example, the socket were a datagram socket, *type* would have the value corresponding to SOCK_DGRAM.

5.10. NS Packet Sequences

The semantics of NS connections demand that the user both be able to look inside the network header associated with any incoming packet and be able to specify what should go in certain fields of an outgoing packet. Using different calls to *setsockopt*, it is possible to indicate whether prototype headers will be associated by the user with each outgoing packet (SO_HEADERS_ON_OUTPUT), to indicate whether the headers received by the system should be delivered to the user (SO_HEADERS_ON_INPUT), or to indicate default information that should be associated with all outgoing packets on a given socket (SO_DEFAULT_HEADERS).

The contents of a SPP header (minus the IDP header) are:

```
struct sphdr {
        u_char  sp_cc;                  /* connection control */
#define SP_SP   0x80                    /* system packet */
#define SP_SA   0x40                    /* send acknowledgement */
#define SP_OB   0x20                    /* attention (out of band data) */
#define SP_EM   0x10                    /* end of message */
        u_char  sp_dt;                  /* datastream type */
        u_short sp_sid;                 /* source connection identifier */
        u_short sp_did;                 /* destination connection identifier */
        u_short sp_seq;                 /* sequence number */
        u_short sp_ack;                 /* acknowledge number */
        u_short sp_alo;                 /* allocation number */
};
```

Here, the items of interest are the *datastream type* and the *connection control* fields. The semantics of the datastream type are defined by the application(s) in question; the value of this field is, by default, zero, but it can be used to indicate things such as Xerox's Bulk Data Transfer Protocol (in which case it is set to one). The connection control field is a mask of the flags defined just below it. The user may set or clear the end-of-message bit to indicate that a given message is the last of a given substream type, or may set/clear the attention bit as an alternate way to indicate that a packet should be sent out-of-band. As an example, to associate prototype headers with outgoing SPP packets, consider:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/sp.h>
 ...
struct sockaddr_ns sns, to;
int s, on = 1;
struct databuf {
      struct sphdr proto_spp;     /* prototype header */
      char buf[534];              /* max. possible data by Xerox std. */
} buf;
 ...
s = socket(AF_NS, SOCK_SEQPACKET, 0);
 ...
bind(s, (struct sockaddr *) &sns, sizeof (sns));
setsockopt(s, NSPROTO_SPP, SO_HEADERS_ON_OUTPUT, &on, sizeof(on));
 ...
buf.proto_spp.sp_dt = 1;   /* bulk data */
buf.proto_spp.sp_cc = SP_EM;  /* end-of-message */
strcpy(buf.buf, "hello world\n");
sendto(s, (char *) &buf, sizeof(struct sphdr) + strlen("hello world\n"),
   (struct sockaddr *) &to, sizeof(to));
 ...
```

Note that one must be careful when writing headers; if the prototype header is not written with the data with which it is to be associated, the kernel will treat the first few bytes of the data as the header, with unpredictable results. To turn off the above association, and to indicate that packet headers received by the system should be passed up to the user, one might use:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/sp.h>
...
struct sockaddr sns;
int s, on = 1, off = 0;
...
s = socket(AF_NS, SOCK_SEQPACKET, 0);
...
bind(s, (struct sockaddr *) &sns, sizeof (sns));
setsockopt(s, NSPROTO_SPP, SO_HEADERS_ON_OUTPUT, &off, sizeof(off));
setsockopt(s, NSPROTO_SPP, SO_HEADERS_ON_INPUT, &on, sizeof(on));
...
```

Output is handled somewhat differently in the IDP world. The header of an IDP-level packet looks like:

```
struct idp {
        u_short         idp_sum;        /* Checksum */
        u_short         idp_len;        /* Length, in bytes, including header */
        u_char          idp_tc;         /* Transport Control (i.e., hop count) */
        u_char          idp_pt;         /* Packet Type (i.e., level 2 protocol) */
        struct ns_addr  idp_dna;        /* Destination Network Address */
        struct ns_addr  idp_sna;        /* Source Network Address */
};
```

The primary field of interest in an IDP header is the *packet type* field. The standard values for this field are (as defined in *<netns/ns.h>*):

```
#define NSPROTO_RI      1       /* Routing Information */
#define NSPROTO_ECHO    2       /* Echo Protocol */
#define NSPROTO_ERROR   3       /* Error Protocol */
#define NSPROTO_PE      4       /* Packet Exchange */
#define NSPROTO_SPP     5       /* Sequenced Packet */
```

For SPP connections, the contents of this field are automatically set to NSPROTO_SPP; for IDP packets, this value defaults to zero, which means "unknown".

Setting the value of that field with SO_DEFAULT_HEADERS is easy:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/idp.h>
 ...
struct sockaddr sns;
struct idp proto_idp;        /* prototype header */
int s, on = 1;
 ...
s = socket(AF_NS, SOCK_DGRAM, 0);
 ...
bind(s, (struct sockaddr *) &sns, sizeof (sns));
proto_idp.idp_pt = NSPROTO_PE;  /* packet exchange */
setsockopt(s, NSPROTO_IDP, SO_DEFAULT_HEADERS, (char *) &proto_idp,
   sizeof(proto_idp));
 ...
```

Using SO_HEADERS_ON_OUTPUT is somewhat more difficult. When SO_HEADERS_ON_OUTPUT is turned on for an IDP socket, the socket becomes (for all intents and purposes) a raw socket. In this case, all the fields of the prototype header (except the length and checksum fields, which are computed by the kernel) must be filled in correctly in order for the socket to send and receive data in a sensible manner. To be more specific, the source address must be set to that of the host sending the data; the destination address must be set to that of the host for whom the data is intended; the packet type must be set to whatever value is desired; and the hopcount must be set to some reasonable value (almost always zero). It should also be noted that simply sending data using *write* will not work unless a *connect* or *sendto* call is used, in spite of the fact that it is the destination address in the prototype header that is used, not the one given in either of those calls. For almost all IDP applications , using SO_DEFAULT_HEADERS is easier and more desirable than writing headers.

5.11. Three-way Handshake

The semantics of SPP connections indicates that a three-way handshake, involving changes in the datastream type, should — but is not absolutely required to — take place before a SPP connection is closed. Almost all SPP connections are "well-behaved" in this manner; when communicating with any process, it is best to assume that the three-way handshake is required unless it is known for certain that it is not required. In a three-way close, the closing process indicates that it wishes to close the connection by sending a zero-length packet with end-of-message set and with datastream type 254. The other side of the connection indicates that it is OK to close by sending a zero-length packet with end-of-message set and datastream type 255. Finally, the closing process replies with a zero-length packet with substream type 255; at this point, the connection is considered closed. The following code fragments are simplified examples of how one might handle this three-way handshake at the user level; in the future, support for this type of close will probably be provided as part of the C library or as part of the kernel. The first code fragment below illustrates how a process might handle three-way handshake if it sees that the process it is communicating with wants to close the connection:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/sp.h>
 ...
#ifndef SPPSST_END
#define SPPSST_END 254
#define SPPSST_ENDREPLY 255
#endif
struct sphdr proto_sp;
int s;
 ...
read(s, buf, BUFSIZE);
if (((struct sphdr *)buf)->sp_dt == SPPSST_END) {
        /*
         * SPPSST_END indicates that the other side wants to
         * close.
         */
        proto_sp.sp_dt = SPPSST_ENDREPLY;
        proto_sp.sp_cc = SP_EM;
        setsockopt(s, NSPROTO_SPP, SO_DEFAULT_HEADERS, (char *)&proto_sp,
            sizeof(proto_sp));
        write(s, buf, 0);
        /*
         * Write a zero-length packet with datastream type = SPPSST_ENDREPLY
         * to indicate that the close is OK with us.  The packet that we
         * don't see (because we don't look for it) is another packet
         * from the other side of the connection, with SPPSST_ENDREPLY
         * on it it, too.  Once that packet is sent, the connection is
         * considered closed; note that we really ought to retransmit
         * the close for some time if we do not get a reply.
         */
        close(s);
}
 ...
```

To indicate to another process that we would like to close the connection, the following code would suffice:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/sp.h>
 ...
#ifndef SPPSST_END
#define SPPSST_END 254
#define SPPSST_ENDREPLY 255
#endif
struct sphdr proto_sp;
int s;
 ...
proto_sp.sp_dt = SPPSST_END;
proto_sp.sp_cc = SP_EM;
setsockopt(s, NSPROTO_SPP, SO_DEFAULT_HEADERS, (char *)&proto_sp,
   sizeof(proto_sp));
write(s, buf, 0); /* send the end request */
proto_sp.sp_dt = SPPSST_ENDREPLY;
setsockopt(s, NSPROTO_SPP, SO_DEFAULT_HEADERS, (char *)&proto_sp,
   sizeof(proto_sp));
/*
 * We assume (perhaps unwisely)
 * that the other side will send the
 * ENDREPLY, so we'll just send our final ENDREPLY
 * as if we'd seen theirs already.
 */
write(s, buf, 0);
close(s);
 ...
```

5.12. Packet Exchange

The Xerox standard protocols include a protocol that is both reliable and datagram-oriented. This protocol is known as Packet Exchange (PEX or PE) and, like SPP, is layered on top of IDP. PEX is important for a number of things: Courier remote procedure calls may be expedited through the use of PEX, and many Xerox servers are located by doing a PEX "BroadcastForServers" operation. Although there is no implementation of PEX in the kernel, it may be simulated at the user level with some clever coding and the use of one peculiar *getsockopt*. A PEX packet looks like:

```
/*
 * The packet-exchange header shown here is not defined
 * as part of any of the system include files.
 */
struct pex {
        struct idp  p_idp;              /* idp header */
        u_short     ph_id[2];           /* unique transaction ID for pex */
        u_short     ph_client;          /* client type field for pex */
};
```

The *ph_id* field is used to hold a "unique id" that is used in duplicate suppression; the *ph_client* field indicates the PEX client type (similar to the packet type field in the IDP header). PEX reliability stems from the fact that it is an idempotent ("I send a packet to you, you send a packet to me") protocol. Processes on each side of the connection may use the unique id to determine if they have seen a given packet before (the unique id field differs on each packet sent) so that duplicates may be detected, and to indicate which message a given packet is in response to. If a packet with a given

unique id is sent and no response is received in a given amount of time, the packet is retransmitted until it is decided that no response will ever be received. To simulate PEX, one must be able to generate unique ids -- something that is hard to do at the user level with any real guarantee that the id is really unique. Therefore, a means (via *getsockopt*) has been provided for getting unique ids from the kernel. The following code fragment indicates how to get a unique id:

```
long uniqueid;
int s, idsize = sizeof(uniqueid);
 ...
s = socket(AF_NS, SOCK_DGRAM, 0);
 ...
/* get id from the kernel -- only on IDP sockets */
getsockopt(s, NSPROTO_PE, SO_SEQNO, (char *)&uniqueid, &idsize);
 ...
```

The retransmission and duplicate suppression code required to simulate PEX fully is left as an exercise for the reader.

5.13. Inetd

One of the daemons provided with 4.3BSD is *inetd*, the so called "internet super-server." *Inetd* is invoked at boot time, and determines from the file */etc/inetd.conf* the servers for which it is to listen. Once this information has been read and a pristine environment created, *inetd* proceeds to create one socket for each service it is to listen for, binding the appropriate port number to each socket.

Inetd then performs a *select* on all these sockets for read availability, waiting for somebody wishing a connection to the service corresponding to that socket. *Inetd* then performs an *accept* on the socket in question, *fork*s, *dup*s the new socket to file descriptors 0 and 1 (stdin and stdout), closes other open file descriptors, and *exec*s the appropriate server.

Servers making use of *inetd* are considerably simplified, as *inetd* takes care of the majority of the IPC work required in establishing a connection. The server invoked by *inetd* expects the socket connected to its client on file descriptors 0 and 1, and may immediately perform any operations such as *read*, *write*, *send*, or *recv*. Indeed, servers may use buffered I/O as provided by the "stdio" conventions, as long as as they remember to use *fflush* when appropriate.

One call which may be of interest to individuals writing servers under *inetd* is the *getpeername* call, which returns the address of the peer (process) connected on the other end of the socket. For example, to log the Internet address in "dot notation" (e.g., "128.32.0.4") of a client connected to a server under *inetd*, the following code might be used:

```
struct sockaddr_in name;
int namelen = sizeof (name);
 ...
if (getpeername(0, (struct sockaddr *)&name, &namelen) < 0) {
        syslog(LOG_ERR, "getpeername: %m");
        exit(1);
} else
        syslog(LOG_INFO, "Connection from %s", inet_ntoa(name.sin_addr));
 ...
```

While the *getpeername* call is especially useful when writing programs to run with *inetd*, it can be used under other circumstances. Be warned, however, that *getpeername* will fail on UNIX domain sockets.

Lint, a C Program Checker

S. C. Johnson

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Lint is a command which examines C source programs, detecting a number of bugs and obscurities. It enforces the type rules of C more strictly than the C compilers. It may also be used to enforce a number of portability restrictions involved in moving programs between different machines and/or operating systems. Another option detects a number of wasteful, or error prone, constructions which nevertheless are, strictly speaking, legal.

Lint accepts multiple input files and library specifications, and checks them for consistency.

The separation of function between *lint* and the C compilers has both historical and practical rationale. The compilers turn C programs into executable files rapidly and efficiently. This is possible in part because the compilers do not do sophisticated type checking, especially between separately compiled programs. *Lint* takes a more global, leisurely view of the program, looking much more carefully at the compatibilities.

This document discusses the use of *lint*, gives an overview of the implementation, and gives some hints on the writing of machine independent C code.

Introduction and Usage

Suppose there are two C[1] source files, *file1.*c and *file2.c*, which are ordinarily compiled and loaded together. Then the command

```
lint file1.c file2.c
```

produces messages describing inconsistencies and inefficiencies in the programs. The program enforces the typing rules of C more strictly than the C compilers (for both historical and practical reasons) enforce them. The command

```
lint -p file1.c file2.c
```

will produce, in addition to the above messages, additional messages which relate to the portability of the programs to other operating systems and machines. Replacing the **-p** by **-h** will produce messages about various error-prone or wasteful constructions which, strictly speaking, are not bugs. Saying **-hp** gets the whole works.

The next several sections describe the major messages; the document closes with sections discussing the implementation and giving suggestions for writing portable C. An appendix gives a summary of the *lint* options.

A Word About Philosophy

Many of the facts which *lint* needs may be impossible to discover. For example, whether a given function in a program ever gets called may depend on the input data. Deciding whether *exit* is ever called is equivalent to solving the famous "halting problem," known to be recursively

undecidable.

Thus, most of the *lint* algorithms are a compromise. If a function is never mentioned, it can never be called. If a function is mentioned, *lint* assumes it can be called; this is not necessarily so, but in practice is quite reasonable.

Lint tries to give information with a high degree of relevance. Messages of the form "*xxx* might be a bug" are easy to generate, but are acceptable only in proportion to the fraction of real bugs they uncover. If this fraction of real bugs is too small, the messages lose their credibility and serve merely to clutter up the output, obscuring the more important messages.

Keeping these issues in mind, we now consider in more detail the classes of messages which *lint* produces.

Unused Variables and Functions

As sets of programs evolve and develop, previously used variables and arguments to functions may become unused; it is not uncommon for external variables, or even entire functions, to become unnecessary, and yet not be removed from the source. These "errors of commission" rarely cause working programs to fail, but they are a source of inefficiency, and make programs harder to understand and change. Moreover, information about such unused variables and functions can occasionally serve to discover bugs; if a function does a necessary job, and is never called, something is wrong!

Lint complains about variables and functions which are defined but not otherwise mentioned. An exception is variables which are declared through explicit **extern** statements but are never referenced; thus the statement

```
extern float sin();
```

will evoke no comment if *sin* is never used. Note that this agrees with the semantics of the C compiler. In some cases, these unused external declarations might be of some interest; they can be discovered by adding the **-x** flag to the *lint* invocation.

Certain styles of programming require many functions to be written with similar interfaces; frequently, some of the arguments may be unused in many of the calls. The **-v** option is available to suppress the printing of complaints about unused arguments. When **-v** is in effect, no messages are produced about unused arguments except for those arguments which are unused and also declared as register arguments; this can be considered an active (and preventable) waste of the register resources of the machine.

There is one case where information about unused, or undefined, variables is more distracting than helpful. This is when *lint* is applied to some, but not all, files out of a collection which are to be loaded together. In this case, many of the functions and variables defined may not be used, and, conversely, many functions and variables defined elsewhere may be used. The **-u** flag may be used to suppress the spurious messages which might otherwise appear.

Set/Used Information

Lint attempts to detect cases where a variable is used before it is set. This is very difficult to do well; many algorithms take a good deal of time and space, and still produce messages about perfectly valid programs. *Lint* detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a "use," since the actual use may occur at any later time, in a data dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement, since the true flow of control need not be discovered. It does mean that *lint* can complain about some programs which are legal, but these programs would probably be considered bad on stylistic grounds (e.g. might contain at least two **goto**'s). Because static and external variables are initialized to 0, no meaningful information can be discovered about their uses. The algorithm deals correctly, however, with initialized automatic variables, and variables which are used in the expression which first sets them.

The set/used information also permits recognition of those local variables which are set and never used; these form a frequent source of inefficiencies, and may also be symptomatic of bugs.

Flow of Control

Lint attempts to detect unreachable portions of the programs which it processes. It will complain about unlabeled statements immediately following **goto**, **break**, **continue**, or **return** statements. An attempt is made to detect loops which can never be left at the bottom, detecting the special cases **while**(1) and **for**(;;) as infinite loops. *Lint* also complains about loops which cannot be entered at the top; some valid programs may have such loops, but at best they are bad style, at worst bugs.

Lint has an important area of blindness in the flow of control algorithm: it has no way of detecting functions which are called and never return. Thus, a call to *exit* may cause unreachable code which *lint* does not detect; the most serious effects of this are in the determination of returned function values (see the next section).

One form of unreachable statement is not usually complained about by *lint;* a **break** statement that cannot be reached causes no message. Programs generated by *yacc*,[2] and especially *lex*,[3] may have literally hundreds of unreachable **break** statements. The **-O** flag in the C compiler will often eliminate the resulting object code inefficiency. Thus, these unreached statements are of little importance, there is typically nothing the user can do about them, and the resulting messages would clutter up the *lint* output. If these messages are desired, *lint* can be invoked with the **-b** option.

Function Values

Sometimes functions return values which are never used; sometimes programs incorrectly use function "values" which have never been returned. *Lint* addresses this problem in a number of ways.

Locally, within a function definition, the appearance of both

```
return( expr );
```

and

```
return ;
```

statements is cause for alarm; *lint* will give the message

```
function name contains return(e) and return
```

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

```
f ( a ) {
        if ( a ) return ( 3 );
        g ();
        }
```

Notice that, if *a* tests false, *f* will call *g* and then return with no defined return value; this will trigger a complaint from *lint*. If *g*, like *exit*, never returns, the message will still be produced when in fact nothing is wrong.

In practice, some potentially serious bugs have been discovered by this feature; it also accounts for a substantial fraction of the "noise" messages produced by *lint*.

On a global scale, *lint* detects cases where a function returns a value, but this value is sometimes, or always, unused. When the value is always unused, it may constitute an inefficiency in the function definition. When the value is sometimes unused, it may represent bad style (e.g., not testing for error conditions).

The dual problem, using a function value when the function does not return one, is also detected. This is a serious problem. Amazingly, this bug has been observed on a couple of occasions in "working" programs; the desired function value just happened to have been computed in the function return register!

Type Checking

Lint enforces the type checking rules of C more strictly than the compilers do. The additional checking is in four major areas: across certain binary operators and implied assignments, at the structure selection operators, between the definition and uses of functions, and in the use of enumerations.

There are a number of operators which have an implied balancing between types of the operands. The assignment, conditional (?:), and relational operators have this property; the argument of a **return** statement, and expressions used in initialization also suffer similar conversions. In these operations, **char**, **short**, **int**, **long**, **unsigned**, **float**, and **double** types may be freely intermixed. The types of pointers must agree exactly, except that arrays of *x*'s can, of course, be intermixed with pointers to *x*'s.

The type checking rules also require that, in structure references, the left operand of the —> be a pointer to structure, the left operand of the . be a structure, and the right operand of these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types **float** and **double** may be freely matched, as may the types **char**, **short**, **int**, and **unsigned**. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts.

With enumerations, checks are made that enumeration variables or members are not mixed with other types, or other enumerations, and that the only operations applied are =, initialization, ==, !=, and function arguments and return values.

Type Casts

The type cast feature in C was introduced largely as an aid to producing more portable programs. Consider the assignment

```
p = 1 ;
```

where *p* is a character pointer. *Lint* will quite rightly complain. Now, consider the assignment

```
p = (char *)1 ;
```

in which a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong motivation for doing this, and has clearly signaled his intentions. It seems harsh for *lint* to continue to complain about this. On the other hand, if this code is moved to another machine, such code should be looked at carefully. The -c flag controls the printing of comments about casts. When -c is in effect, casts are treated as though they were assignments subject to complaint; otherwise, all legal casts are passed without comment, no matter how strange the type mixing seems to be.

Nonportable Character Use

On the PDP-11, characters are signed quantities, with a range from -128 to 127. On most of the other C implementations, characters take on only positive values. Thus, *lint* will flag certain comparisons and assignments as being illegal or nonportable. For example, the fragment

```
char c;
    ...
if( (c = getchar()) < 0 ) ....
```

works on the PDP-11, but will fail on machines where characters always take on positive values. The real solution is to declare *c* an integer, since *getchar* is actually returning integer values. In any case, *lint* will say "nonportable character comparison".

A similar issue arises with bitfields; when assignments of constant values are made to bitfields, the field may be too small to hold the value. This is especially true because on some machines bitfields are considered as signed quantities. While it may seem unintuitive to consider that a two bit

field declared of type **int** cannot hold the value 3, the problem disappears if the bitfield is declared to have type **unsigned.**

Assignments of longs to ints

Bugs may arise from the assignment of **long** to an **int**, which loses accuracy. This may happen in programs which have been incompletely converted to use **typedefs.** When a **typedef** variable is changed from **int** to **long**, the program can stop working because some intermediate results may be assigned to **ints**, losing accuracy. Since there are a number of legitimate reasons for assigning **longs** to **ints**, the detection of these assignments is enabled by the **-a** flag.

Strange Constructions

Several perfectly legal, but somewhat strange, constructions are flagged by *lint;* the messages hopefully encourage better code quality, clearer style, and may even point out bugs. The **-h** flag is used to enable these checks. For example, in the statement

```
*p++ ;
```

the * does nothing; this provokes the message "null effect" from *lint*. The program fragment

```
unsigned x ;
if( x < 0 ) ...
```

is clearly somewhat strange; the test will never succeed. Similarly, the test

```
if( x > 0 ) ...
```

is equivalent to

```
if( x != 0 )
```

which may not be the intended action. *Lint* will say "degenerate unsigned comparison" in these cases. If one says

```
if( 1 != 0 ) ....
```

lint will report "constant in conditional context", since the comparison of 1 with 0 gives a constant result.

Another construction detected by *lint* involves operator precedence. Bugs which arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statements

```
if( x&077 == 0 ) ...
```

or

```
x<<2 + 40
```

probably do not do what was intended. The best solution is to parenthesize such expressions, and *lint* encourages this by an appropriate message.

Finally, when the **-h** flag is in force *lint* complains about variables which are redeclared in inner blocks in a way that conflicts with their use in outer blocks. This is legal, but is considered by many (including the author) to be bad style, usually unnecessary, and frequently a bug.

Ancient History

There are several forms of older syntax which are being officially discouraged. These fall into two classes, assignment operators and initialization.

The older forms of assignment operators (e.g., =+, =-, . . .) could cause ambiguous expressions, such as

```
a =-1 ;
```

which could be taken as either

```
a =- 1 ;
```

or

```
a = -1 ;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer, and preferred operators (+=, -=, etc.) have no such ambiguities. To spur the abandonment of the older forms, *lint* complains about these old fashioned operators.

A similar issue arises with initialization. The older language allowed

```
int x 1 ;
```

to initialize x to 1. This also caused syntactic difficulties: for example,

```
int x (-1) ;
```

looks somewhat like the beginning of a function declaration:

```
int x (y) { ...
```

and the compiler must read a fair ways past x in order to sure what the declaration really is.. Again, the problem is even more perplexing when the initializer involves a macro. The current syntax places an equals sign between the variable and the initializer:

```
int x = -1 ;
```

This is free of any possible syntactic ambiguity.

Pointer Alignment

Certain pointer assignments may be reasonable on some machines, and illegal on others, due entirely to alignment restrictions. For example, on the PDP-11, it is reasonable to assign integer pointers to double pointers, since double precision values may begin on any integer boundary. On the Honeywell 6000, double precision values must begin on even word boundaries; thus, not all such assignments make sense. *Lint* tries to detect cases where pointers are assigned to other pointers, and such alignment problems might arise. The message "possible pointer alignment problem" results from this situation whenever either the **-p** or **-h** flags are in effect.

Multiple Uses and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine dependent. For example, on machines (like the PDP-11) in which the stack runs backwards, function arguments will probably be best evaluated from right-to-left; on machines with a stack running forward, left-to-right seems most attractive. Function calls embedded as arguments of other functions may or may not be treated similarly to ordinary arguments. Similar issues arise with other operators which have side effects, such as the assignment operators and the increment and decrement operators.

In order that the efficiency of C on a particular machine not be unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler, and, in fact, the various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect, and also used elsewhere in the same expression, the result is explicitly undefined.

Lint checks for the important special case where a simple scalar variable is affected. For example, the statement

```
a[i] = b[i++] ;
```

will draw the complaint:

warning: *i* evaluation order undefined

Implementation

Lint consists of two programs and a driver. The first program is a version of the Portable C Compiler[4,5] which is the basis of the IBM 370, Honeywell 6000, and Interdata 8/32 C compilers. This compiler does lexical and syntax analysis on the input text, constructs and maintains symbol tables, and builds trees for expressions. Instead of writing an intermediate file which is passed to a code generator, as the other compilers do, *lint* produces an intermediate file which consists of lines of ascii text. Each line contains an external variable name, an encoding of the context in which it was seen (use, definition, declaration, etc.), a type specifier, and a source file name and line number. The information about variables local to a function or file is collected by accessing the symbol table, and examining the expression trees.

Comments about local problems are produced as detected. The information about external names is collected onto an intermediate file. After all the source files and library descriptions have been collected, the intermediate file is sorted to bring all information collected about a given external name together. The second, rather small, program then reads the lines from the intermediate file and compares all of the definitions, declarations, and uses for consistency.

The driver controls this process, and is also responsible for making the options available to both passes of *lint*.

Portability

C on the Honeywell and IBM systems is used, in part, to write system code for the host operating system. This means that the implementation of C tends to follow local conventions rather than adhere strictly to UNIX† system conventions. Despite these differences, many C programs have been successfully moved to GCOS and the various IBM installations with little effort. This section describes some of the differences between the implementations, and discusses the *lint* features which encourage portability.

Uninitialized external variables are treated differently in different implementations of C. Suppose two files both contain a declaration without initialization, such as

```
int a ;
```

outside of any function. The UNIX loader will resolve these declarations, and cause only a single word of storage to be set aside for *a*. Under the GCOS and IBM implementations, this is not feasible (for various stupid reasons!) so each such declaration causes a word of storage to be set aside and called *a*. When loading or library editing takes place, this causes fatal conflicts which prevent the proper operation of the program. If *lint* is invoked with the **-p** flag, it will detect such multiple definitions.

A related difficulty comes from the amount of information retained about external names during the loading process. On the UNIX system, externally known names have seven significant characters, with the upper/lower case distinction kept. On the IBM systems, there are eight significant characters, but the case distinction is lost. On GCOS, there are only six characters, of a single case. This leads to situations where programs run on the UNIX system, but encounter loader problems on the IBM or GCOS systems. *Lint* **-p** causes all external symbols to be mapped to one case and truncated to six characters, providing a worst-case analysis.

A number of differences arise in the area of character handling: characters in the UNIX system are eight bit ascii, while they are eight bit ebcdic on the IBM, and nine bit ascii on GCOS. Moreover, character strings go from high to low bit positions ("left to right") on GCOS and IBM, and low to high ("right to left") on the PDP-11. This means that code attempting to construct strings out of

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

character constants, or attempting to use characters as indices into arrays, must be looked at with great suspicion. *Lint* is of little help here, except to flag multi-character character constants.

Of course, the word sizes are different! This causes less trouble than might be expected, at least when moving from the UNIX system (16 bit words) to the IBM (32 bits) or GCOS (36 bits). The main problems are likely to arise in shifting or masking. C now supports a bit-field facility, which can be used to write much of this code in a reasonably portable way. Frequently, portability of such code can be enhanced by slight rearrangements in coding style. Many of the incompatibilities seem to have the flavor of writing

```
x &= 0177700 ;
```

to clear the low order six bits of *x*. This suffices on the PDP-11, but fails badly on GCOS and IBM. If the bit field feature cannot be used, the same effect can be obtained by writing

```
x &= ~ 077 ;
```

which will work on all these machines.

The right shift operator is arithmetic shift on the PDP-11, and logical shift on most other machines. To obtain a logical shift on all machines, the left operand can be typed **unsigned.** Characters are considered signed integers on the PDP-11, and unsigned on the other machines. This persistence of the sign bit may be reasonably considered a bug in the PDP-11 hardware which has infiltrated itself into the C language. If there were a good way to discover the programs which would be affected, C could be changed; in any case, *lint* is no help here.

The above discussion may have made the problem of portability seem bigger than it in fact is. The issues involved here are rarely subtle or mysterious, at least to the implementor of the program, although they can involve some work to straighten out. The most serious bar to the portability of UNIX system utilities has been the inability to mimic essential UNIX system functions on the other systems. The inability to seek to a random character position in a text file, or to establish a pipe between processes, has involved far more rewriting and debugging than any of the differences in C compilers. On the other hand, *lint* has been very helpful in moving the UNIX operating system and associated utility programs to other machines.

Shutting Lint Up

There are occasions when the programmer is smarter than *lint*. There may be valid reasons for "illegal" type casts, functions with a variable number of arguments, etc. Moreover, as specified above, the flow of control information produced by *lint* often has blind spots, causing occasional spurious messages about perfectly reasonable programs. Thus, some way of communicating with *lint*, typically to shut it up, is desirable.

The form which this mechanism should take is not at all clear. New keywords would require current and old compilers to recognize these keywords, if only to ignore them. This has both philosophical and practical problems. New preprocessor syntax suffers from similar problems.

What was finally done was to cause a number of words to be recognized by *lint* when they were embedded in comments. This required minimal preprocessor changes; the preprocessor just had to agree to pass comments through to its output, instead of deleting them as had been previously done. Thus, *lint* directives are invisible to the compilers, and the effect on systems with the older preprocessors is merely that the *lint* directives don't work.

The first directive is concerned with flow of control information; if a particular place in the program cannot be reached, but this is not apparent to *lint*, this can be asserted by the directive

```
/* NOTREACHED */
```

at the appropriate spot in the program. Similarly, if it is desired to turn off strict type checking for the next expression, the directive

```
/* NOSTRICT */
```

can be used; the situation reverts to the previous default after the next expression. The –v flag can be turned on for one function by the directive

/* ARGSUSED */

Complaints about variable number of arguments in calls to a function can be turned off by the directive

/* VARARGS */

preceding the function definition. In some cases, it is desirable to check the first several arguments, and leave the later arguments unchecked. This can be done by following the VARARGS keyword immediately with a digit giving the number of arguments which should be checked; thus,

/* VARARGS2 */

will cause the first two arguments to be checked, the others unchecked. Finally, the directive

/* LINTLIBRARY */

at the head of a file identifies this file as a library declaration file; this topic is worth a section by itself.

Library Declaration Files

Lint accepts certain library directives, such as

-ly

and tests the source files for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library directives. These files all begin with the directive

/* LINTLIBRARY */

which is followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function return type, whether the dummy function returns a value, and the number and types of arguments to the function. The VARARGS and ARGSUSED directives can be used to specify features of the library functions.

Lint library files are processed almost exactly like ordinary source files. The only difference is that functions which are defined on a library file, but are not used on a source file, draw no complaints. *Lint* does not simulate a full library search algorithm, and complains if the source files contain a redefinition of a library routine (this is a feature!).

By default, *lint* checks the programs it is given against a standard library file, which contains descriptions of the programs which are normally loaded when a C program is run. When the **-p** flag is in effect, another file is checked containing descriptions of the standard I/O library routines which are expected to be portable across various machines. The **-n** flag can be used to suppress all library checking.

Bugs, etc.

Lint was a difficult program to write, partially because it is closely connected with matters of programming style, and partially because users usually don't notice bugs which cause *lint* to miss errors which it should have caught. (By contrast, if *lint* incorrectly complains about something that is correct, the programmer reports that immediately!)

A number of areas remain to be further developed. The checking of structures and arrays is rather inadequate; size incompatibilities go unchecked, and no attempt is made to match up structure and union declarations across files. Some stricter checking of the use of the **typedef** is clearly desirable, but what checking is appropriate, and how to carry it out, is still to be determined.

Lint shares the preprocessor with the C compiler. At some point it may be appropriate for a special version of the preprocessor to be constructed which checks for things such as unused macro

definitions, macro arguments which have side effects which are not expanded at all, or are expanded more than once, etc.

The central problem with *lint* is the packaging of the information which it collects. There are many options which serve only to turn off, or slightly modify, certain features. There are pressures to add even more of these options.

In conclusion, it appears that the general notion of having two programs is a good one. The compiler concentrates on quickly and accurately turning the program text into bits which can be run; *lint* concentrates on issues of portability, style, and efficiency. *Lint* can afford to be wrong, since incorrectness and over-conservatism are merely annoying, not fatal. The compiler can be fast since it knows that *lint* will cover its flanks. Finally, the programmer can concentrate at one stage of the programming process solely on the algorithms, data structures, and correctness of the program, and then later retrofit, with the aid of *lint*, the desirable properties of universality and portability.

References

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
2. S. C. Johnson, "Yacc — Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey, July 1975. Reprinted as PS1:15 in *UNIX Programmer's Manual,* Usenix Association, (1986).
3. M. E. Lesk, "Lex — A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey, October 1975. Reprinted as PS1:16 in *UNIX Programmer's Manual,* Usenix Association, (1986).
4. S. C. Johnson and D. M. Ritchie, "UNIX Time-Sharing System: Portability of C Programs and the UNIX System," *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 2021-2048, 1978.
5. S. C. Johnson, "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages,* pp. 97-104, January 1978.

Appendix: Current Lint Options

The command currently has the form

lint [–options] files... library-descriptors...

The options are

h Perform heuristic checks

p Perform portability checks

v Don't report unused arguments

u Don't report unused or undefined externals

b Report unreachable **break** statements.

x Report unused external declarations

a Report assignments of **long** to **int** or shorter.

c Complain about questionable casts

n No library checking is done

s Same as **h** (for historical reasons)

PS1 10

A Tutorial Introduction to ADB

J. F. Maranzano

S. R. Bourne

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Debugging tools generally provide a wealth of information about the inner workings of programs. These tools have been available on UNIX† to allow users to examine "core" files that result from aborted programs. A new debugging program, ADB, provides enhanced capabilities to examine "core" and other program files in a variety of formats, run programs with embedded breakpoints and patch files.

ADB is an indispensable but complex tool for debugging crashed systems and/or programs. This document provides an introduction to ADB with examples of its use. It explains the various formatting options, techniques for debugging C programs, examples of printing file system information and patching.

1. Introduction

ADB is a new debugging program that is available on UNIX. It provides capabilities to look at "core" files resulting from aborted programs, print output in a variety of formats, patch files, and run programs with embedded breakpoints. This document provides examples of the more useful features of ADB. The reader is expected to be familiar with the basic commands on UNIX with the C language, and with References 1, 2 and 3.

2. A Quick Survey

2.1. Invocation

ADB is invoked as:

adb objfile corefile

where *objfile* is an executable UNIX file and *corefile* is a core image file. Many times this will look like:

adb a.out core

or more simply:

adb

where the defaults are *a.out* and *core* respectively. The filename minus (–) means ignore this argument as in:

adb – core

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

ADB has requests for examining locations in either file. The ? request examines the contents of *objfile*, the / request examines the *corefile*. The general form of these requests is:

address ? format

or

address / format

2.2. Current Address

ADB maintains a current address, called dot, similar in function to the current pointer in the UNIX editor. When an address is entered, the current address is set to that location, so that:

0126?i

sets dot to octal 126 and prints the instruction at that address. The request:

.,10/d

prints 10 decimal numbers starting at dot. Dot ends up referring to the address of the last item printed. When used with the ? or / requests, the current address can be advanced by typing newline; it can be decremented by typing ^.

Addresses are represented by expressions. Expressions are made up from decimal, octal, and hexadecimal integers, and symbols from the program under test. These may be combined with the operators +, –, *, % (integer division), & (bitwise and), | (bitwise inclusive or), # (round up to the next multiple), and ~ (not). (All arithmetic within ADB is 32 bits.) When typing a symbolic address for a C program, the user can type *name* or *_name;* ADB will recognize both forms.

2.3. Formats

To print data, a user specifies a collection of letters and characters that describe the format of the printout. Formats are "remembered" in the sense that typing a request without one will cause the new printout to appear in the previous format. The following are the most commonly used format letters.

b	**one byte in octal**
c	**one byte as a character**
o	**one word in octal**
d	**one word in decimal**
f	**two words in floating point**
i	**PDP 11 instruction**
s	**a null terminated character string**
a	**the value of dot**
u	**one word as unsigned integer**
n	**print a newline**
r	**print a blank space**
^	**backup dot**

(Format letters are also available for "long" values, for example, 'D' for long decimal, and 'F' for double floating point.) For other formats see the ADB manual.

2.4. General Request Meanings

The general form of a request is:

address,count command modifier

which sets 'dot' to *address* and executes the command *count* times.

The following table illustrates some general ADB command meanings:

Command	Meaning
?	**Print contents from *a.out* file**
/	**Print contents from *core* file**
=	**Print value of "dot"**
:	**Breakpoint control**
$	**Miscellaneous requests**
;	**Request separator**
!	**Escape to shell**

ADB catches signals, so a user cannot use a quit signal to exit from ADB. The request $q or $Q (or cntl-D) must be used to exit from ADB.

3. Debugging C Programs

3.1. Debugging A Core Image

Consider the C program in Figure 1. The program is used to illustrate a common error made by C programmers. The object of the program is to change the lower case "t" to upper case in the string pointed to by *charp* and then write the character string to the file indicated by argument 1. The bug shown is that the character "T" is stored in the pointer *charp* instead of the string pointed to by *charp.* Executing the program produces a core file because of an out of bounds memory reference.

ADB is invoked by:

adb a.out core

The first debugging request:

$c

is used to give a C backtrace through the subroutines called. As shown in Figure 2 only one function (*main*) was called and the arguments *argc* and *argv* have octal values 02 and 0177762 respectively. Both of these values look reasonable; 02 = two arguments, 0177762 = address on stack of parameter vector.
The next request:

$C

is used to give a C backtrace plus an interpretation of all the local variables in each function and their values in octal. The value of the variable *cc* looks incorrect since *cc* was declared as a character.

The next request:

$r

prints out the registers including the program counter and an interpretation of the instruction at that location.

The request:

$e

prints out the values of all external variables.

A map exists for each file handled by ADB. The map for the *a.out* file is referenced by ? whereas the map for *core* file is referenced by /. Furthermore, a good rule of thumb is to use ? for instructions and / for data when looking at programs. To print out information about the maps type:

$m

This produces a report of the contents of the maps. More about these maps later.

In our example, it is useful to see the contents of the string pointed to by *charp.* This is done by:

***charp/s**

which says use *charp* as a pointer in the *core* file and print the information as a character string. This printout clearly shows that the character buffer was incorrectly overwritten and helps identify the error. Printing the locations around *charp* shows that the buffer is unchanged but that the pointer is destroyed. Using ADB similarly, we could print information about the arguments to a function. The request:

main.argc/d

prints the decimal *core* image value of the argument *argc* in the function *main.*
The request:

***main.argv,3/o**

prints the octal values of the three consecutive cells pointed to by *argv* in the function *main.* Note that these values are the addresses of the arguments to main. Therefore:

0177770/s

prints the ASCII value of the first argument. Another way to print this value would have been

***"/s**

The " means ditto which remembers the last address typed, in this case *main.argc* ; the * instructs ADB to use the address field of the *core* file as a pointer.

The request:

.=o

prints the current address (not its contents) in octal which has been set to the address of the first argument. The current address, dot, is used by ADB to "remember" its current location. It allows the user to reference locations relative to the current address, for example:

.-10/d

3.2. Multiple Functions

Consider the C program illustrated in Figure 3. This program calls functions *f, g,* and *h* until the stack is exhausted and a core image is produced.

Again you can enter the debugger via:

adb

which assumes the names *a.out* and *core* for the executable file and core image file respectively. The request:

$c

will fill a page of backtrace references to *f, g,* and *h.* Figure 4 shows an abbreviated list (typing *DEL* will terminate the output and bring you back to ADB request level).

The request:

,5$C

prints the five most recent activations.

Notice that each function (*f,g,h*) has a counter of the number of times it was called.

The request:

fcnt/d

prints the decimal value of the counter for the function *f.* Similarly *gcnt* and *hcnt* could be printed. To print the value of an automatic variable, for example the decimal value of *x* in the last call of the

function *h*, type:

h.x/d

It is currently not possible in the exported version to print stack frames other than the most recent activation of a function. Therefore, a user can print everything with $C or the occurrence of a variable in the most recent call of a function. It is possible with the $C request, however, to print the stack frame starting at some address as **address$C**.

3.3. Setting Breakpoints

Consider the C program in Figure 5. This program, which changes tabs into blanks, is adapted from *Software Tools* by Kernighan and Plauger, pp. 18-27.

We will run this program under the control of ADB (see Figure 6a) by:

adb a.out -

Breakpoints are set in the program as:

address:b [request]

The requests:

settab+4:b
fopen+4:b
getc+4:b
tabpos+4:b

set breakpoints at the start of these functions. C does not generate statement labels. Therefore it is currently not possible to plant breakpoints at locations other than function entry points without a knowledge of the code generated by the C compiler. The above addresses are entered as **symbol+4** so that they will appear in any C backtrace since the first instruction of each function is a call to the C save routine (*csv*). Note that some of the functions are from the C library.

To print the location of breakpoints one types:

$b

The display indicates a *count* field. A breakpoint is bypassed *count -1* times before causing a stop. The *command* field indicates the ADB requests to be executed each time the breakpoint is encountered. In our example no *command* fields are present.

By displaying the original instructions at the function *settab* we see that the breakpoint is set after the jsr to the C save routine. We can display the instructions using the ADB request:

settab,5?ia

This request displays five instructions starting at *settab* with the addresses of each location displayed. Another variation is:

settab,5?i

which displays the instructions with only the starting address.

Notice that we accessed the addresses from the *a.out* file with the ? command. In general when asking for a printout of multiple items, ADB will advance the current address the number of bytes necessary to satisfy the request; in the above example five instructions were displayed and the current address was advanced 18 (decimal) bytes.

To run the program one simply types:

:r

To delete a breakpoint, for instance the entry to the function *settab*, one types:

settab+4:d

To continue execution of the program from the breakpoint type:

```
:c
```

Once the program has stopped (in this case at the breakpoint for *fopen),* ADB requests can be used to display the contents of memory. For example:

```
$C
```

to display a stack trace, or:

```
tabs,3/8o
```

to print three lines of 8 locations each from the array called *tabs.* By this time (at location *fopen)* in the C program, *settab* has been called and should have set a one in every eighth location of *tabs.*

3.4. Advanced Breakpoint Usage

We continue execution of the program with:

```
:c
```

See Figure 6b. *Getc* is called three times and the contents of the variable *c* in the function *main* are displayed each time. The single character on the left hand edge is the output from the C program. On the third occurrence of *getc* the program stops. We can look at the full buffer of characters by typing:

```
ibuf+6/20c
```

When we continue the program with:

```
:c
```

we hit our first breakpoint at *tabpos* since there is a tab following the "This" word of the data.

Several breakpoints of *tabpos* will occur until the program has changed the tab into equivalent blanks. Since we feel that *tabpos* is working, we can remove the breakpoint at that location by:

```
tabpos+4:d
```

If the program is continued with:

```
:c
```

it resumes normal execution after ADB prints the message

```
a.out:running
```

The UNIX quit and interrupt signals act on ADB itself rather than on the program being debugged. If such a signal occurs then the program being debugged is stopped and control is returned to ADB. The signal is saved by ADB and is passed on to the test program if:

```
:c
```

is typed. This can be useful when testing interrupt handling routines. The signal is not passed on to the test program if:

```
:c 0
```

is typed.

Now let us reset the breakpoint at *settab* and display the instructions located there when we reach the breakpoint. This is accomplished by:

```
settab+4:b settab,5?ia *
```

It is also possible to execute the ADB requests for each occurrence of the breakpoint but only stop

* Owing to a bug in early versions of ADB (including the version distributed in Generic 3 UNIX) these

after the third occurrence by typing:

```
getc+4,3:b  main.c?C  *
```

This request will print the local variable *c* in the function *main* at each occurrence of the breakpoint. The semicolon is used to separate multiple ADB requests on a single line.

Warning: setting a breakpoint causes the value of dot to be changed; executing the program under ADB does not change dot. Therefore:

```
settab+4:b  .,5?ia
fopen+4:b
```

will print the last thing dot was set to (in the example *fopen+4*) *not* the current location (*settab+4*) at which the program is executing.

A breakpoint can be overwritten without first deleting the old breakpoint. For example:

```
settab+4:b  settab,5?ia; ptab/o  *
```

could be entered after typing the above requests.

Now the display of breakpoints:

```
$b
```

shows the above request for the *settab* breakpoint. When the breakpoint at *settab* is encountered the ADB requests are executed. Note that the location at *settab+4* has been changed to plant the breakpoint; all the other locations match their original value.

Using the functions, *f, g* and *h* shown in Figure 3, we can follow the execution of each function by planting non-stopping breakpoints. We call ADB with the executable program of Figure 3 as follows:

```
adb ex3 -
```

Suppose we enter the following breakpoints:

```
h+4:b     hcnt/d; h.hi/; h.hr/
g+4:b     gcnt/d; g.gi/; g.gr/
f+4:b     fcnt/d; f.fi/; f.fr/
:r
```

Each request line indicates that the variables are printed in decimal (by the specification **d**). Since the format is not changed, the **d** can be left off all but the first request.

The output in Figure 7 illustrates two points. First, the ADB requests in the breakpoint line are not examined until the program under test is run. That means any errors in those ADB requests is not detected until run time. At the location of the error ADB stops running the program.

The second point is the way ADB handles register variables. ADB uses the symbol table to address variables. Register variables, like *f.fr* above, have pointers to uninitialized places on the stack. Therefore the message "symbol not found".

Another way of getting at the data in this example is to print the variables used in the call as:

```
f+4:b     fcnt/d; f.a/; f.b/; f.fi/
g+4:b     gcnt/d; g.p/; g.q/; g.gi/
:c
```

---

statements must be written as:

```
settab+4:b        settab,5?ia;0
getc+4,3:b        main.c?C;0
settab+4:b        settab,5?ia; ptab/o;0
```

Note that **;0** will set dot to zero and stop at the breakpoint.

The operator / was used instead of ? to read values from the *core* file. The output for each function, as shown in Figure 7, has the same format. For the function *f*, for example, it shows the name and value of the *external* variable *fcnt*. It also shows the address on the stack and value of the variables *a*, *b* and *fi*.

Notice that the addresses on the stack will continue to decrease until no address space is left for program execution at which time (after many pages of output) the program under test aborts. A display with names would be produced by requests like the following:

f+4:b fcnt/d; f.a/"a="d; f.b/"b="d; f.fi/"fi="d

In this format the quoted string is printed literally and the **d** produces a decimal display of the variables. The results are shown in Figure 7.

3.5. Other Breakpoint Facilities

- Arguments and change of standard input and output are passed to a program as:

  :r arg1 arg2 ... <infile >outfile

  This request kills any existing program under test and starts the *a.out* afresh.
- The program being debugged can be single stepped by:

  :s

  If necessary, this request will start up the program being debugged and stop after executing the first instruction.
- ADB allows a program to be entered at a specific address by typing:

  address:r

- The count field can be used to skip the first *n* breakpoints as:

  ,n:r

  The request:

  ,n:c

  may also be used for skipping the first *n* breakpoints when continuing a program.

- A program can be continued at an address different from the breakpoint by:

  address:c

- The program being debugged runs as a separate process and can be killed by:

  :k

4. Maps

UNIX supports several executable file formats. These are used to tell the loader how to load the program file. File type 407 is the most common and is generated by a C compiler invocation such as **cc pgm.c**. A 410 file is produced by a C compiler command of the form **cc -n pgm.c**, whereas a 411 file is produced by **cc -i pgm.c**. ADB interprets these different file formats and provides access to the different segments through a set of maps (see Figure 8). To print the maps type:

$m

In 407 files, both text (instructions) and data are intermixed. This makes it impossible for ADB to differentiate data from instructions and some of the printed symbolic addresses look incorrect; for example, printing data addresses as offsets from routines.

In 410 files (shared text), the instructions are separated from data and **?*** accesses the data part of the *a.out* file. The **?*** request tells ADB to use the second part of the map in the *a.out* file. Accessing data in the *core* file shows the data after it was modified by the execution of the program. Notice also that the data segment may have grown during program execution.

In 411 files (separated I & D space), the instructions and data are also separated. However, in this case, since data is mapped through a separate set of segmentation registers, the base of the data segment is also relative to address zero. In this case since the addresses overlap it is necessary to use the **?*** operator to access the data space of the *a.out* file. In both 410 and 411 files the corresponding core file does not contain the program text.

Figure 9 shows the display of three maps for the same program linked as a 407, 410, 411 respectively. The b, e, and f fields are used by ADB to map addresses into file addresses. The "f1" field is the length of the header at the beginning of the file (020 bytes for an *a.out* file and 02000 bytes for a *core* file). The "f2" field is the displacement from the beginning of the file to the data. For a 407 file with mixed text and data this is the same as the length of the header; for 410 and 411 files this is the length of the header plus the size of the text portion.

The "b" and "e" fields are the starting and ending locations for a segment. Given an address, A, the location in the file (either *a.out* or *core*) is calculated as:

b1≤A≤e1 ⇒ file address = (A–b1)+f1
b2≤A≤e2 ⇒ file address = (A–b2)+f2

A user can access locations by using the ADB defined variables. The **$v** request prints the variables initialized by ADB:

b	**base address of data segment**
d	**length of the data segment**
s	**length of the stack**
t	**length of the text**
m	**execution type (407,410,411)**

In Figure 9 those variables not present are zero. Use can be made of these variables by expressions such as:

<b

in the address field. Similarly the value of the variable can be changed by an assignment request such as:

02000>b

that sets **b** to octal 2000. These variables are useful to know if the file under examination is an executable or *core* image file.

ADB reads the header of the *core* image file to find the values for these variables. If the second file specified does not seem to be a *core* file, or if it is missing then the header of the executable file is used instead.

5. Advanced Usage

It is possible with ADB to combine formatting requests to provide elaborate displays. Below are several examples.

5.1. Formatted dump

The line:

<b,–1/4o4^8Cn

prints 4 octal words followed by their ASCII interpretation from the data space of the core image file. Broken down, the various request pieces mean:

<b The base address of the data segment.

<b,-1 Print from the base address to the end of file. A negative count is used here and elsewhere to loop indefinitely or until some error condition (like end of file) is detected.

The format **4o4^8Cn** is broken down as follows:

4o Print 4 octal locations.

4^ Backup the current address 4 locations (to the original start of the field).

8C Print 8 consecutive characters using an escape convention; each character in the range 0 to 037 is printed as @ followed by the corresponding character in the range 0140 to 0177. An @ is printed as @@.

n Print a newline.

The request:

<b,<d/4o4^8Cn

could have been used instead to allow the printing to stop at the end of the data segment (<d provides the data segment size in bytes).

The formatting requests can be combined with ADB's ability to read in a script to produce a core image dump script. ADB is invoked as:

adb a.out core < dump

to read in a script file, *dump,* of requests. An example of such a script is:

```
120$w
4095$s
$v
=3n
$m
=3n"C Stack Backtrace"
$C
=3n"C External Variables"
$e
=3n"Registers"
$r
0$s
=3n"Data Segment"
<b,-1/8ona
```

The request **120$w** sets the width of the output to 120 characters (normally, the width is 80 characters). ADB attempts to print addresses as:

symbol + offset

The request **4095$s** increases the maximum permissible offset to the nearest symbolic address from 255 (default) to 4095. The request = can be used to print literal strings. Thus, headings are provided in this *dump* program with requests of the form:

=3n"C Stack Backtrace"

that spaces three lines and prints the literal string. The request $v prints all non-zero ADB variables (see Figure 8). The request **0$s** sets the maximum offset for symbol matches to zero thus suppressing

the printing of symbolic labels in favor of octal values. Note that this is only done for the printing of the data segment. The request:

```
<b,-1/8ona
```

prints a dump from the base of the data segment to the end of file with an octal address field and eight octal numbers per line.

Figure 11 shows the results of some formatting requests on the C program of Figure 10.

5.2. Directory Dump

As another illustration (Figure 12) consider a set of requests to dump the contents of a directory (which is made up of an integer *inumber* followed by a 14 character name):

```
adb dir -
=n8t"Inum"8t"Name"
0,-1? u8t14cn
```

In this example, the **u** prints the *inumber* as an unsigned decimal integer, the **8t** means that ADB will space to the next multiple of 8 on the output line, and the **14c** prints the 14 character file name.

5.3. Ilist Dump

Similarly the contents of the *ilist* of a file system, (e.g. /dev/src, on UNIX systems distributed by the UNIX Support Group; see UNIX Programmer's Manual Section V) could be dumped with the following set of requests:

```
adb /dev/src -
02000>b
?m <b
<b,-1?"flags"8ton"links,uid,gid"8t3bn",size"8tbrdn"addr"8t8un"times"8t2Y2na
```

In this example the value of the base for the map was changed to 02000 (by saying **?m<b**) since that is the start of an *ilist* within a file system. An artifice (**brd** above) was used to print the 24 bit size field as a byte, a space, and a decimal integer. The last access time and last modify time are printed with the **2Y** operator. Figure 12 shows portions of these requests as applied to a directory and file system.

5.4. Converting values

ADB may be used to convert values from one representation to another. For example:

```
072 = odx
```

will print

```
072        58        #3a
```

which is the octal, decimal and hexadecimal representations of 072 (octal). The format is remembered so that typing subsequent numbers will print them in the given formats. Character values may be converted similarly, for example:

```
'a' = co
```

prints

```
a          0141
```

It may also be used to evaluate expressions but be warned that all binary operators have the same precedence which is lower than that for unary operators.

6. Patching

Patching files with ADB is accomplished with the *write,* **w** or **W**, request (which is not like the *ed* editor write command). This is often used in conjunction with the *locate,* **l** or **L** request. In general, the request syntax for **l** and **w** are similar as follows:

?l value

The request **l** is used to match on two bytes, **L** is used for four bytes. The request **w** is used to write two bytes, whereas **W** writes four bytes. The **value** field in either *locate* or *write* requests is an expression. Therefore, decimal and octal numbers, or character strings are supported.

In order to modify a file, ADB must be called as:

adb –w file1 file2

When called with this option, *file1* and *file2* are created if necessary and opened for both reading and writing.

For example, consider the C program shown in Figure 10. We can change the word "This" to "The " in the executable file for this program, *ex7*, by using the following requests:

```
adb –w ex7 –
?l 'Th'
?W 'The '
```

The request **?l** starts at dot and stops at the first match of "Th" having set dot to the address of the location found. Note the use of **?** to write to the *a.out* file. The form **?*** would have been used for a 411 file.

More frequently the request will be typed as:

?l 'Th'; ?s

and locates the first occurrence of "Th" and print the entire string. Execution of this ADB request will set dot to the address of the "Th" characters.

As another example of the utility of the patching facility, consider a C program that has an internal logic flag. The flag could be set by the user through ADB and the program run. For example:

```
adb a.out –
:s arg1 arg2
flag/w 1
:c
```

The **:s** request is normally used to single step through a process or start a process in single step mode. In this case it starts *a.out* as a subprocess with arguments **arg1** and **arg2**. If there is a subprocess running ADB writes to it rather than to the file so the **w** request causes *flag* to be changed in the memory of the subprocess.

7. Anomalies

Below is a list of some strange things that users should be aware of.

1. Function calls and arguments are put on the stack by the C save routine. Putting breakpoints at the entry point to routines means that the function appears not to have been called when the breakpoint occurs.
2. When printing addresses, ADB uses either text or data symbols from the *a.out* file. This sometimes causes unexpected symbol names to be printed with data (e.g. *savr5+022*). This does not happen if **?** is used for text (instructions) and **/** for data.
3. ADB cannot handle C register variables in the most recently activated function.

8. Acknowledgements

The authors are grateful for the thoughtful comments on how to organize this document from R. B. Brandt, E. N. Pinson and B. A. Tague. D. M. Ritchie made the system changes necessary to accommodate tracing within ADB. He also participated in discussions during the writing of ADB. His earlier work with DB and CDB led to many of the features found in ADB.

9. References

1. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," CACM, July, 1974.
2. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, 1978.
3. K. Thompson and D. M. Ritchie, UNIX Programmer's Manual - 7th Edition, 1978.
4. B. W. Kernighan and P. J. Plauger, *Software Tools,* Addison-Wesley, 1976.

Figure 1: C program with pointer bug

```
struct buf {
        int fildes;
        int nleft;
        char *nextp;
        char buff[512];
        }bb;
struct buf *obuf;

char *charp "this is a sentence.";

main(argc,argv)
int argc;
char **argv;
{
        char    cc;

        if(argc < 2) {
                printf("Input file missing\n");
                exit(8);
        }

        if((fcreat(argv[1],obuf)) < 0){
                printf("%s : not found\n", argv[1]);
                exit(8);
        }
        charp = 'T';
printf("debug 1 %s\n",charp);
        while(cc= *charp++)
                putc(cc,obuf);
        fflush(obuf);
}
```

Figure 2: ADB output for C program of Figure 1

```
adb a.out core
$c
~main(02,0177762)
$C
~main(02,0177762)
        argc:       02
        argv:       0177762
        cc:         02124
$r
ps      0170010
pc      0204    ~main+0152
sp      0177740
r5      0177752
r4      01
r3      0
r2      0
r1      0
r0      0124
~main+0152:     mov     _obuf,(sp)
$e
savr5:      0
_obuf:      0
_charp:     0124
_errno:     0
_fout:      0
$m
text map    `ex1'
b1 = 0              e1   = 02360          f1 = 020
b2 = 0              e2   = 02360          f2 = 020
data map    `core1'
b1 = 0              e1   = 03500          f1 = 02000
b2 = 0175400        e2   = 0200000              f2 = 05500
*charp/s
0124:           TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTLx    Nh@x &_
~
charp/s
_charp:         T

_charp+02:      this is a sentence.

_charp+026:     Input file missing
main.argc/d
0177756:        2
*main.argv/3o
0177762:        01777001777760177777
0177770/s
0177770:        a.out
*main.argv/3o
0177762:        01777001777760177777
*"/s
0177770:        a.out
 .=o
                0177770
 .-10/d
0177756:        2
$q
```

Figure 3: Multiple function C program for stack trace illustration

```
int     fcnt,gcnt,hcnt;
h(x,y)
{
        int hi; register int hr;
        hi = x+1;
        hr = x-y+1;
        hcnt++ ;
        hj:
        f(hr,hi);
}

g(p,q)
{
        int gi; register int gr;
        gi = q-p;
        gr = q-p+1;
        gcnt++ ;
        gj:
        h(gr,gi);
}

f(a,b)
{
        int fi; register int fr;
        fi = a+2*b;
        fr = a+b;
        fcnt++ ;
        fj:
        g(fr,fi);
}

main()
{
        f(1,1);
}
```

Figure 4: ADB output for C program of Figure 3

```
adb
$c
~h(04452,04451)
~g(04453,011124)
~f(02,04451)
~h(04450,04447)
~g(04451,011120)
~f(02,04447)
~h(04446,04445)
~g(04447,011114)
~f(02,04445)
~h(04444,04443)
HIT DEL KEY
adb
,5$C
~h(04452,04451)
        x:      04452
        y:      04451
        hi:     ?
~g(04453,011124)
        p:      04453
        q:      011124
        gi:     04451
        gr:     ?
~f(02,04451)
        a:      02
        b:      04451
        fi:     011124
        fr:     04453
~h(04450,04447)
        x:      04450
        y:      04447
        hi:     04451
        hr:     02
~g(04451,011120)
        p:      04451
        q:      011120
        gi:     04447
        gr:     04450
fcnt/d
_fcnt:          1173
gcnt/d
_gcnt:          1173
hcnt/d
_hcnt:          1172
h.x/d
022004:         2346
$q
```

Figure 5: C program to decode tabs

```
#define MAXLINE     80
#define YES         1
#define NO          0
#define TABSP       8

char    input[] "data";
char    ibuf[518];
int     tabs[MAXLINE];

main()
{
        int col, *ptab;
        char c;

        ptab = tabs;
        settab(ptab);       /*Set initial tab stops */
        col = 1;
        if(fopen(input,ibuf) < 0) {
                printf("%s : not found\n",input);
                exit(8);
        }
        while((c = getc(ibuf)) != -1) {
                switch(c) {
                        case '\t': /* TAB */
                                while(tabpos(col) != YES) {
                                        putchar(' ');       /* put BLANK */
                                        col++ ;
                                }
                                break;
                        case '\n':/*NEWLINE */
                                putchar('\n');
                                col = 1;
                                break;
                        default:
                                putchar(c);
                                col++ ;
                }
        }
}
/* Tabpos return YES if col is a tab stop */
tabpos(col)
int col;
{
        if(col > MAXLINE)
                return(YES);
        else
                return(tabs[col]);
}
/* Settab - Set initial tab stops */
settab(tabp)
int *tabp;
{
        int i;

        for(i = 0; i<= MAXLINE; i++)
                (i%TABSP) ? (tabs[i] = NO) : (tabs[i] = YES);
}
```

Figure 6a: ADB output for C program of Figure 5

```
adb a.out -
settab+4:b
fopen+4:b
getc+4:b
tabpos+4:b
$b
breakpoints
count    bkpt            command
1        ~tabpos+04
1        _getc+04
1        _fopen+04
1        ~settab+04
settab,5?ia
~settab:        jsr     r5,csv
~settab+04:     tst     -(sp)
~settab+06:     clr     0177770(r5)
~settab+012:    cmp     $0120,0177770(r5)
~settab+020:    blt     ~settab+076
~settab+022:
settab,5?i
~settab:        jsr     r5,csv
                tst     -(sp)
                clr     0177770(r5)
                cmp     $0120,0177770(r5)
                blt     ~settab+076
:r
a.out: running
breakpoint      ~settab+04:     tst     -(sp)
settab+4:d
:c
a.out: running
breakpoint      _fopen+04:      mov     04(r5),nulstr+012
$C
_fopen(02302,02472)
~main(01,0177770)
        col:    01
        c:      0
        ptab:   03500
tabs,3/8o
03500:          01      0       0       0       0       0       0       0
                01      0       0       0       0       0       0       0
                01      0       0       0       0       0       0       0
```

Figure 6b: ADB output for C program of Figure 5

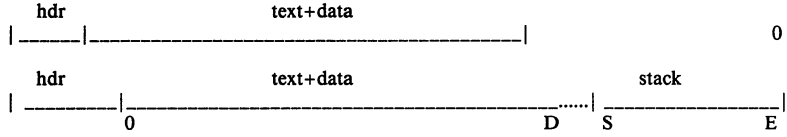
```
:c
a.out: running
breakpoint      _getc+04:       mov     04(r5),r1
ibuf+6/20c
__cleanu+0202:          This    is      a test  of
:c
a.out: running
breakpoint      ~tabpos+04:     cmp     $0120,04(r5)
tabpos+4:d
settab+4:b  settab,5?ia
settab+4:b  settab,5?ia;  0
getc+4,3:b  main.c?C;  0
settab+4:b  settab,5?ia;  ptab/o;  0
$b
breakpoints
count   bkpt            command
1       ~tabpos+04
3       _getc+04        main.c?C;0
1       _fopen+04
1       ~settab+04      settab,5?ia;ptab?o;0
~settab:        jsr     r5,csv
~settab+04:     bpt
~settab+06:     clr     0177770(r5)
~settab+012:    cmp     $0120,0177770(r5)
~settab+020:    blt     ~settab+076
~settab+022:
0177766:        0177770
0177744:        @
T0177744:       T
h0177744:       h
i0177744:       i
s0177744:       s
```

Figure 7: ADB output for C program with breakpoints

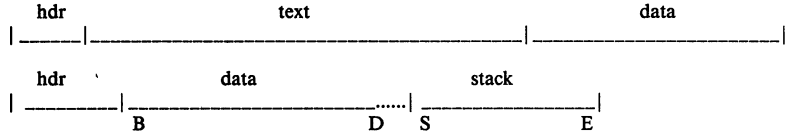
```
adb ex3 -
h+4:b hcnt/d; h.hi/; h.hr/
g+4:b gcnt/d; g.gi/; g.gr/
f+4:b fcnt/d; f.fi/; f.fr/
:r
ex3: running
_fcnt:      0
0177732:    214
symbol not found
f+4:b fcnt/d; f.a/; f.b/; f.fi/
g+4:b gcnt/d; g.p/; g.q/; g.gi/
h+4:b hcnt/d; h.x/; h.y/; h.hi/
:c
ex3: running
_fcnt:      0
0177746:    1
0177750:    1
0177732:    214
_gcnt:      0
0177726:    2
0177730:    3
0177712:    214
_hcnt:          0
0177706:    2
0177710:    1
0177672:    214
_fcnt:      1
0177666:    2
0177670:    3
0177652:    214
_gcnt:      1
0177646:    5
0177650:    8
0177632:    214
HIT DEL
f+4:b fcnt/d; f.a/"a = "d; f.b/"b = "d; f.fi/"fi = "d
g+4:b gcnt/d; g.p/"p = "d; g.q/"q = "d; g.gi/"gi = "d
h+4:b hcnt/d; h.x/"x = "d; h.y/"h = "d; h.hi/"hi = "d
:r
ex3: running
_fcnt:      0
0177746:    a = 1
0177750:    b = 1
0177732:    fi = 214
_gcnt:      0
0177726:    p = 2
0177730:    q = 3
0177712:    gi = 214
_hcnt:          0
0177706:    x = 2
0177710:    y = 1
0177672:    hi = 214
_fcnt:      1
0177666:    a = 2
0177670:    b = 3
0177652:    fi = 214
HIT DEL
$q
```

Figure 8: ADB address maps

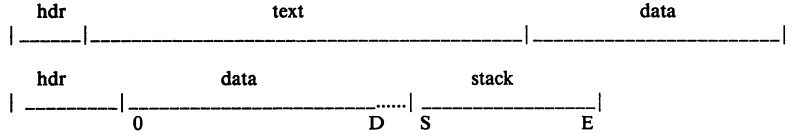
407 files



410 files (shared text)



411 files (separated I and D space)



The following *adb* variables are set.

		407	410	411
b	base of data	0	B	0
d	length of data	D	D–B	D
s	length of stack	S	S	S
t	length of text	0	T	T

Figure 9: ADB output for maps

```
adb map407 core407
$m
text map   `map407'
b1 = 0              e1    = 0256          f1 = 020
b2 = 0              e2    = 0256          f2 = 020
data map   `core407'
b1 = 0              e1    = 0300          f1 = 02000
b2 = 0175400        e2    = 0200000       f2 = 02300
$v
variables
d = 0300
m = 0407
s = 02400
$q

adb map410 core410
$m
text map   `map410'
b1 = 0              e1    = 0200          f1 = 020
b2 = 020000         e2    = 020116   f2 = 0220
data map   `core410'
b1 = 020000         e1    = 020200   f1 = 02000
b2 = 0175400        e2    = 0200000       f2 = 02200
$v
variables
b = 020000
d = 0200
m = 0410
s = 02400
t = 0200
$q

adb map411 core411
$m
text map   `map411'
b1 = 0              e1    = 0200          f1 = 020
b2 = 0              e2    = 0116          f2 = 0220
data map   `core411'
b1 = 0              e1    = 0200          f1 = 02000
b2 = 0175400        e2    = 0200000       f2 = 02200
$v
variables
d = 0200
m = 0411
s = 02400
t = 0200
$q
```

Figure 10: Simple C program for illustrating formatting and patching

```
char    str1[]   "This is a character string";
int     one      1;
int     number 456;
long    lnum     1234;
float   fpt      1.25;
char    str2[]   "This is the second character string";
main()
{
        one = 2;
}
```

Figure 11: ADB output illustrating fancy formats

```
adb map410 core410
<b,-1/8ona
020000:        0    064124    071551    064440    020163    020141    064143071141
_str1+016:061541    062564    020162    072163    064562    063556    0     02

_number:
_number: 0710 0     02322     040240    0     064124    071551    064440

_str2+06: 020163    064164    020145    062563    067543    062156    061440060550

_str2+026:060562    072143    071145    071440    071164    067151    0147 0

savr5+02: 0    0    0    0    0    0    0    0

<b,20/4o4^8Cn
020000:        0    064124    071551    064440    @'@'This i
          020163    020141    064143    071141    s a char
          061541    062564    020162    072163    acter st
          064562    063556    0    02   ring@'@'@b@'

_number: 0710 0     02322     040240    H@a@'@'R@d @@
          0    064124    071551    064440    @'@'This i
          020163    064164    020145    062563    s the se
          067543    062156    061440    060550    cond cha
          060562    072143    071145    071440    racter s
          071164    067151    0147 0    tring@'@'@'
          0    0    0    0    @'@'@'@'@'@'@'
          0    0    0    0    @'@'@'@'@'@'@'
data address not found
<b,20/4o4^8t8cna
020000:        0    064124    071551    064440         This i
_str1+06: 020163    020141    064143    071141         s a char
_str1+016:061541    062564    020162    072163         acter st
_str1+026:064562    063556    0    02        ring
_number:
_number: 0710 0     02322     040240         HR
_fpt+02:  0    064124    071551    064440         This i
_str2+06: 020163    064164    020145    062563         s the se
_str2+016:067543    062156    061440    060550         cond cha
_str2+026:060562    072143    071145    071440         racter    s
_str2+036:071164    067151    0147 0         tring
savr5+02: 0    0    0    0
savr5+012:     0    0    0    0
data address not found
<b,10/2b8t^2cn
020000:        0    0

_str1:              0124 0150      Th
          0151 0163      is
          040  0151       i
          0163 040        s
          0141 040        a
          0143 0150      ch
          0141 0162      ar
          0141 0143      ac
          0164 0145      te
$Q
```

Figure 12: Directory and inode dumps

```
adb dir -
=nt"Inode"t"Name"
0,-1?ut14cn

          Inode     Name
0:        652  .
          82   ..
          5971 cap.c
          5323 cap
          0    pp



adb /dev/src -
02000>b
?m<b
new map    `/dev/src'
b1 = 02000        e1     = 0100000000   f1 = 0
b2 = 0            e2     = 0            f2 = 0
$v
variables
b = 02000
<b,-1?"flags"8ton"links,uid,gid"8t3bn"size"8tbrdn"addr"8t8un"times"8t2Y2na
02000:          flags 073145
          links,uid,gid     0163 0164 0141
          size  0162 10356
          addr 28770     8236 25956     27766     25455     8236 25956     25206
          times1976 Feb 5 08:34:56  1975 Dec 28 10:55:15

02040:          flags 024555
          links,uid,gid     012  0163 0164
          size  0162 25461
          addr 8308 30050     8294 25130     15216     26890     29806     10784
          times1976 Aug 17 12:16:51      1976 Aug 17 12:16:51

02100:          flags 05173
          links,uid,gid     011  0162 0145
          size  0147 29545
          addr 25972     8306 28265     8308 25642     15216     2314 25970
          times1977 Apr 2 08:58:01  1977 Feb 5 10:21:44
```

ADB Summary

Command Summary

a) formatted printing

? *format*	print from *a.out* file according to *format*
/ *format*	print from *core* file according to *format*
= *format*	print the value of *dot*
?w expr	write expression into *a.out* file
/w expr	write expression into *core* file
?l expr	locate expression in *a.out* file

b) breakpoint and program control

:b	set breakpoint at *dot*
:c	continue running program
:d	delete breakpoint
:k	kill the program being debugged
:r	run *a.out* file under ADB control
:s	single step

c) miscellaneous printing

$b	print current breakpoints
$c	C stack trace
$e	external variables
$f	floating registers
$m	print ADB segment maps
$q	exit from ADB
$r	general registers
$s	set offset for symbol match
$v	print ADB variables
$w	set output line width

d) calling the shell

!	call *shell* to read rest of line

e) assignment to variables

>*name*	assign dot to variable or register *name*

Format Summary

a	the value of dot
b	one byte in octal
c	one byte as a character
d	one word in decimal
f	two words in floating point
i	PDP 11 instruction
o	one word in octal
n	print a newline
r	print a blank space
s	a null terminated character string
*n*t	move to next *n* space tab
u	one word as unsigned integer
x	hexadecimal
Y	date
^	backup dot
"..."	print string

Expression Summary

a) expression components

decimal integer	e.g. 256
octal integer	e.g. 0277
hexadecimal	e.g. #ff
symbols	e.g. flag _main main.argc
variables	e.g. <b
registers	e.g. <pc <r0
(expression)	expression grouping

b) dyadic operators

+	add
-	subtract
*	multiply
%	integer division
&	bitwise and
|	bitwise or
#	round up to the next multiple

c) monadic operators

~	not
*	contents of location
-	integer negate

Debugging with dbx

Bill Tuthill

Sun Microsystems, Inc.
2550 Garcia Avenue

Kevin J. Dunlap

Computer Systems Research Group
University of California
Berkeley, CA 94720

PS 11 1

Introduction

This short paper discusses *dbx*, a symbolic debugger that is vastly superior to *adb*. It may be as good as the debuggers you remember from those non- UNIX† systems you worked on before. The advantage of symbolic debuggers is that they allow you to work with the same names (symbols) as in your source code.

Like *adb*, *dbx* is interactive and line-oriented, but *dbx* is a source-level rather than an assembly-level debugger. It allows you to determine where a program crashed, to view the values of variables and expressions, to set breakpoints in the code, and to run and trace a program. Source code may be in C, Fortran, or Pascal.

Mark Linton wrote *dbx* as his master's thesis at UC Berkeley. Along with Eric Schmidt's Berknet, *dbx* is among the most successful master's theses done on UNIX. Since *dbx* required changes to the symbol tables generated by the various compilers, you need to compile programs for debugging with the *-g* flag. For example, C programs should be compiled as follows:

% cc -g *program*.c -o *program*

Programs compiled with the *-g* option have good symbol tables, while programs compiled without *-g* have old-style symbol tables intended for *adb*. Stripped programs have no symbol tables at all. Invoke the debugger as follows, where *program* is the pathname of the executable file that dumped core:

% dbx *program*

The core image should be in the working directory; if it isn't, specify its pathname in the argument after the program name. Among the great advances of *dbx* is that it has a help facility; type the *help* request to see a list of possible requests. You can obtain help on any *dbx* request by giving its name as an argument to *help*.

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

Examining Core Dumps

Much of the time, programmers use *dbx* to find out why a program dumped core. As an example, consider the following program *dumpcore.c*, which dereferences a NULL pointer. This is a legal operation on VAX/UNIX, but not on VAX/VMS or on MC68000-based UNIX systems, on one of which this example was run:

```
#include <stdio.h>

#define LIMIT 5

main()          /* print messages and die */
{
    int i;

    for (i = 1; i <= 10 ; i++) {
        printf("Goodbye world! (%d)\n", i);
        dumpcore(i);
    }
    exit(0);
}

int *ip;

dumpcore(lim)       /* dereference NULL pointer */
int lim;
{
    if (lim >= LIMIT)
        *ip = lim;
}
```

The program core dumps because of a segmentation violation or memory fault — on most machines it is illegal to assign to address zero. Once the program has produced a core dump, here's how you can find out why the program died:

```
% dbx dumpcore
dbx version 3.17 of 4/24/86 15:04 (monet.Berkeley.EDU).
Type 'help' for help.
reading symbolic information ...
[using memory image in core]
(dbx) where
dumpcore.dumpcore(lim = 5), line 22 in "dumpcore.c"
main(0x1, 0x7fffe904, 0x7fffe90c), line 11 in "dumpcore.c"
```

The *where* request yields a stack trace. As you can see, the *dumpcore()* routine was called from line 11 of the program, with the argument *lim* equal to 5. You can look at the *dumpcore()* procedure by invoking the *list* request as follows:

```
(dbx) list dumpcore
  18  dumpcore(lim)       /* dereference NULL pointer */
  19  int lim;
  20  {
  21      if (lim >= LIMIT)
  22          *ip = lim;
  23  }
```

We immediately suspect that the program's failure had something to do with **ip*, so we use the *print* request to retrieve the value of the pointer and what it points to:

```
(dbx) print *ip
reference through nil pointer
(dbx) print ip
(nil)
```

This tells us the program has dereferenced a null pointer. It is possible to run the program again from inside the debugger. The first line tells you name of the running program, and successive lines give output from the program:

```
(dbx) run
Goodbye world! (1)
Goodbye world! (2)
Goodbye world! (3)
Goodbye world! (4)
Goodbye world! (5)

Bus error in dumpcore.dumpcore at line 22
  22        *ip = lim;
(dbx) quit
```


In this example the program dies with a Bus error at line 22. This method of running the program does not produce a core dump, but the *where* request will still behave properly, because the debugger is in the same state as if it had just read the core file.

Setting Breakpoints

With *dbx* you can set breakpoints before each line of a program, not just at function and procedure boundaries, as with *adb*. The *stop* request sets a breakpoint. After setting a breakpoint, use the *run* request to execute the program. The *cont* request continues execution from the current stopping point until the program finishes or another breakpoint is encountered. The *step* request executes one source statement, following any function calls. The *next* request executes one source statement, but does not stop inside any function calls. The *status* request lists active breakpoints, while the *delete* request removes them if required.

The *stop* request can take a conditional expression to avoid needless single-stepping. We will use a conditional in our example to make things simpler. Of course you can use *print* and *list* requests at any time during statement stepping if you want to print the value of variables or list lines of source code. This sample session shows a mixture of requests as we verify that the program fails when it tries to assign to **ip*:

```
(dbx) stop at 10 if (i == 5)
[1] if i = 5 { stop } at 10
(dbx) run
Goodbye world! (1)
Goodbye world! (2)
Goodbye world! (3)
Goodbye world! (4)
[1] stopped in main at line 10
  10                printf("Goodbye world! (%d)\n", i);
(dbx) next
Goodbye world! (5)
stopped in main at line 11
  11                dumpcore(i);
(dbx) step
stopped in dumpcore at line 21
  21          if (lim >= LIMIT)
(dbx) step
stopped in dumpcore at line 22
  22                *ip = lim;
(dbx) step
Bus error in dumpcore.dumpcore at line 22
  22        *ip = lim;
```

Running the program with breakpoints assures us that our intuition was correct. We shouldn't be assigning anything to a null pointer — *ip* should have been initialized to point at an object of the proper type. To exit from the debugger, use the *quit* request.

It is possible to set variables from inside *dbx*. The previous breakpoint session, for example, could have gone like this:

```
% dbx dumpcore
dbx version 3.17 of 4/24/86 15:04 (monet.Berkeley.EDU).
Type 'help' for help.
reading symbolic information ...
[using memory image in core]
(dbx) stop at 10
[1] stop at 10
(dbx) run
Running: dumpcore
stopped in main at line 10
  10                printf("Goodbye world! (%d)\n", i);
(dbx) assign i = 5
(dbx) next
Goodbye world! (5)
stopped in main at line 11
  11                dumpcore(i);
(dbx) next
Bus error in dumpcore.dumpcore at line 22
  22        *ip = lim;
```

It is often useful to assign new values to variables to draw conclusions about alternative conditions. We can't fix the bug in this program, however, because there is no declared variable to which *ip* should point.

Conclusion

Expressions in *dbx* are similar to those in C, except that there is a distinction between / (floating-point division) and *div* (integer division), as in Pascal. The table on the following page shows *dbx* requests organized by function:

Like *adb*, *dbx* can disassemble object code. It can also examine object files and print output in various formats; but *dbx* requires the proper symbol tables, so *adb* is more useful to examine arbitrary binary files. The most important thing *adb* can do that *dbx* cannot is to patch binary files — *dbx* has no write option. Despite these shortcomings, *dbx* is much easier to use than *adb*, so it contributes much more to individual programmer productivity.

Acknowledgements

Material presented in this document was first presented in "C Advisor", *Unix Review 4*, 1, pp 78–85. The Regents of the University California expresses their gratitude to Unix Review for allowing them to reprint this document.

This document is a good starting point for a more thorough tutorial. Those with the ambition to expand on this document are encouraged to contact the Computer Systems Research Group at "4bsd-ideas@Berkeley.Edu."

Groups of *dbx* Requests	
	execution and tracing
`run`	execute object file
`cont`	continue execution from where it stopped
`trace`	display tracing information at specified place
`stop`	stop execution at specified place
`status`	display active *trace* and *stop* requests
`delete`	delete specific *trace* or *stop* requests
`catch`	start trapping specified signals
`ignore`	stop trapping specified signals
`step`	execute the next source line, stepping into functions
`next`	execute the next source line, even if it's a function
	displaying data
`print`	print the value of an expression
`whatis`	print the declaration of a given identifier or type
`which`	print outer block associated with identifier
`whereis`	print all symbols matching identifier
`assign`	set the value of a variable
	function and procedure handling
`where`	display active procedures and functions on stack
`down`	move down the stack towards stopping point
`up`	move up the stack towards *main*
`call`	call the named function or procedure
`dump`	display names and values of all local variables
	accessing source files and directories
`edit`	invoke an editor on current source file
`file`	change current source file
`func`	change the current function or procedure
`list`	display lines of source code
`use`	set directory list to search for source files
`/.../`	search down in file to match regular expression
`?...?`	search up in file to match regular expression
	miscellaneous commands
`sh`	pass command line to the shell
`alias`	change *dbx* command name
`help`	explain commands
`source`	read commands from external file
`quit`	exit the debugger

Make — A Program for Maintaining Computer Programs

S. I. Feldman

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

In a programming project, it is easy to lose track of which files need to be reprocessed or recompiled after a change is made in some part of the source. *Make* provides a simple mechanism for maintaining up-to-date versions of programs that result from many operations on a number of files. It is possible to tell *Make* the sequence of commands that create certain files, and the list of files that require other files to be current before the operations can be done. Whenever a change is made in any part of the program, the *Make* command will create the proper files simply, correctly, and with a minimum amount of effort.

The basic operation of *Make* is to find the name of a needed target in the description, ensure that all of the files on which it depends exist and are up to date, and then create the target if it has not been modified since its generators were. The description file really defines the graph of dependencies; *Make* does a depth-first search of this graph to determine what work is really necessary.

Make also provides a simple macro substitution facility and the ability to encapsulate commands in a single file for convenient administration.

Revised April, 1986

Introduction

It is common practice to divide large programs into smaller, more manageable pieces. The pieces may require quite different treatments: some may need to be run through a macro processor, some may need to be processed by a sophisticated program generator (e.g., Yacc[1] or Lex[2]). The outputs of these generators may then have to be compiled with special options and with certain definitions and declarations. The code resulting from these transformations may then need to be loaded together with certain libraries under the control of special options. Related maintenance activities involve running complicated test scripts and installing validated modules. Unfortunately, it is very easy for a programmer to forget which files depend on which others, which files have been modified recently, and the exact sequence of operations needed to make or exercise a new version of the program. After a long editing session, one may easily lose track of which files have been changed and which object modules are still valid, since a change to a declaration can obsolete a dozen other files. Forgetting to compile a routine that has been changed or that uses changed declarations will result in a program that will not work, and a bug that can be very hard to track down. On the other hand, recompiling everything in sight just to be safe is very wasteful.

The program described in this report mechanizes many of the activities of program development and maintenance. If the information on inter-file dependences and command sequences is stored in a file, the simple command

```
make
```

is frequently sufficient to update the interesting files, regardless of the number that have been edited since the last "make". In most cases, the description file is easy to write and changes infrequently. It is usually easier to type the *make* command than to issue even one of the needed operations, so the typical cycle of program development operations becomes

think — edit — *make* — test . . .

Make is most useful for medium-sized programming projects; it does not solve the problems of maintaining multiple source versions or of describing huge programs. *Make* was designed for use on Unix, but a version runs on GCOS.

Basic Features

The basic operation of *make* is to update a target file by ensuring that all of the files on which it depends exist and are up to date, then creating the target if it has not been modified since its dependents were. *Make* does a depth-first search of the graph of dependences. The operation of the command depends on the ability to find the date and time that a file was last modified.

To illustrate, let us consider a simple example: A program named *prog* is made by compiling and loading three C-language files *x.c*, *y.c*, and *z.c* with the *lS* library. By convention, the output of the C compilations will be found in files named *x.o*, *y.o*, and *z.o*. Assume that the files *x.c* and *y.c* share some declarations in a file named *defs*, but that *z.c* does not. That is, *x.c* and *y.c* have the line

```
#include "defs"
```

The following text describes the relationships and operations:

```
prog : x.o y.o z.o
	cc x.o y.o z.o -lS -o prog

x.o y.o : defs
```

If this information were stored in a file named *makefile*, the command

```
make
```

would perform the operations needed to recreate *prog* after any changes had been made to any of the four source files *x.c*, *y.c*, *z.c*, or *defs*.

Make operates using three sources of information: a user-supplied description file (as above), file names and "last-modified" times from the file system, and built-in rules to bridge some of the gaps. In our example, the first line says that *prog* depends on three "*.o*" files. Once these object files are current, the second line describes how to load them to create *prog*. The third line says that *x.o* and *y.o* depend on the file *defs*. From the file system, *make* discovers that there are three "*.c*" files corresponding to the needed "*.o*" files, and uses built-in information on how to generate an object from a source file (*i.e.*, issue a "cc –c" command).

The following long-winded description file is equivalent to the one above, but takes no advantage of *make*'s innate knowledge:

```
prog : x.o y.o z.o
	cc x.o y.o z.o -lS -o prog
x.o : x.c defs
	cc -c x.c
y.o : y.c defs
	cc -c y.c
z.o : z.c
	cc -c z.c
```

If none of the source or object files had changed since the last time *prog* was made, all of the files would be current, and the command

```
make
```

would just announce this fact and stop. If, however, the *defs* file had been edited, *x.c* and *y.c* (but not *z.c*) would be recompiled, and then *prog* would be created from the new "*.o*" files. If only the file *y.c* had changed, only it would be recompiled, but it would still be necessary to reload *prog*.

If no target.name is given on the *make* command line, the first target mentioned in the description is created; otherwise the specified targets are made. The command

```
make x.o
```

would recompile *x.o* if *x.c* or *defs* had changed.

If the file exists after the commands are executed, its time of last modification is used in further decisions; otherwise the current time is used. It is often quite useful to include rules with mnemonic names and commands that do not actually produce a file with that name. These entries can take advantage of *make*'s ability to generate files and substitute macros. Thus, an entry "save" might be included to copy a certain set of files, or an entry "cleanup" might be used to throw away unneeded intermediate files. In other cases one may maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

Make has a simple macro mechanism for substituting in dependency lines and command strings. Macros are defined by command arguments or description file lines with embedded equal signs. A macro is invoked by preceding the name by a dollar sign; macro names longer than one character must be parenthesized. The name of the macro is either the single character after the dollar sign or a name inside parentheses. The following are valid macro invocations:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two invocations are identical. $$ is a dollar sign. All of these macros are assigned values during input, as shown below. Four special macros change values during the execution of the command: $*, $@, $?, and $<. They will be discussed later. The following fragment shows the use:

```
OBJECTS = x.o y.o z.o
LIBES = -lS
prog: $(OBJECTS)
    cc $(OBJECTS) $(LIBES) -o prog
...
```

The command

```
make
```

loads the three object files with the *lS* library. The command

```
make "LIBES= -ll -lS"
```

loads them with both the Lex ("-ll") and the Standard ("-lS") libraries, since macro definitions on the command line override definitions in the description. (It is necessary to quote arguments with embedded blanks in UNIX† commands.)

The following sections detail the form of description files and the command line, and discuss options and built-in rules in more detail.

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

Description Files and Substitutions

A description file contains three types of information: macro definitions, dependency information, and executable commands. There is also a comment convention: all characters after a sharp (#) are ignored, as is the sharp itself. Blank lines and lines beginning with a sharp are totally ignored. If a non-comment line is too long, it can be continued using a backslash. If the last character of a line is a backslash, the backslash, newline, and following blanks and tabs are replaced by a single blank.

A macro definition is a line containing an equal sign not preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign (trailing blanks and tabs are stripped) is assigned the string of characters following the equal sign (leading blanks and tabs are stripped.) The following are valid macro definitions:

```
2 = xyz
abc = -ll -ly -lS
LIBES =
```

The last definition assigns LIBES the null string. A macro that is never explicitly defined has the null string as value. Macro definitions may also appear on the *make* command line (see below).

Other lines give information about target files. The general form of an entry is:

```
target1 [target2 . . .] :[:] [dependent1 . . .] [; commands] [# . . .]
[(tab) commands] [# . . .]
. . .
```

Items inside brackets may be omitted. Targets and dependents are strings of letters, digits, periods, and slashes. (Shell metacharacters "*" and "?" are expanded.) A command is any string of characters not including a sharp (except in quotes) or newline. Commands may appear either after a semicolon on a dependency line or on lines beginning with a tab immediately following a dependency line.

A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type.

1. For the usual single-colon case, at most one of these dependency lines may have a command sequence associated with it. If the target is out of date with any of the dependents on any of the lines, and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise a default creation rule may be invoked.
2. In the double-colon case, a command sequence may be associated with each dependency line; if the target is out of date with any of the files on a particular line, the associated commands are executed. A built-in rule may also be executed. This detailed form is of particular value in updating archive-type files.

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the Shell after substituting for macros. (The printing is suppressed in silent mode or if the command line begins with an @ sign). *Make* normally stops if any command signals an error by returning a non-zero error code. (Errors are ignored if the "–i" flags has been specified on the *make* command line, if the fake target name ".IGNORE" appears in the description file, or if the command string in the description file begins with a hyphen. Some UNIX commands return meaningless status). Because each command line is passed to a separate invocation of the Shell, care must be taken with certain commands (e.g., *cd* and Shell control commands) that have meaning only within a single Shell process; the results are forgotten before the next line is executed.

Before issuing any command, certain macros are set. $@ is set to the name of the file to be "made". $? is set to the string of names that were found to be younger than the target. If the command was generated by an implicit rule (see below), $< is the name of the related file that caused the action, and $* is the prefix shared by the current and the dependent file names.

If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name ".DEFAULT" are used. If there is no such name, *make* prints a

message and stops.

Command Usage

The *make* command takes four kinds of arguments: macro definitions, flags, description file names, and target file names.

make [flags] [macro definitions] [targets]

The following summary of the operation of the command explains how these arguments are interpreted.

First, all macro definition arguments (arguments with embedded equal signs) are analyzed and the assignments made. Command-line macros override corresponding definitions found in the description files.

Next, the flag arguments are examined. The permissible flags are

-i Ignore error codes returned by invoked commands. This mode is entered if the fake target name ".IGNORE" appears in the description file.

-s Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name ".SILENT" appears in the description file.

-r Do not use the built-in rules.

-n No execute mode. Print commands, but do not execute them. Even lines beginning with an "@" sign are printed.

-t Touch the target files (causing them to be up to date) rather than issue the usual commands.

-q Question. The *make* command returns a zero or non-zero status code depending on whether the target file is or is not up to date.

-p Print out the complete set of macro definitions and target descriptions

-d Debug mode. Print out detailed information on files and times examined.

-f Description file name. The next argument is assumed to be the name of a description file. A file name of "-" denotes the standard input. If there are no "-f" arguments, the file named *makefile* or *Makefile* in the current directory is read. The contents of the description files override the built-in rules if they are present).

Finally, the remaining arguments are assumed to be the names of targets to be made; they are done in left to right order. If there are no such arguments, the first name in the description files that does not begin with a period is "made".

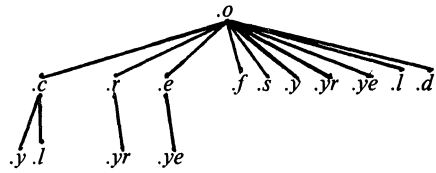
Implicit Rules

The *make* program uses a table of interesting suffixes and a set of transformation rules to supply default dependency information and implied commands. (The Appendix describes these tables and means of overriding them.) The default suffix list is:

.o	Object file
.c	C source file
.e	Efl source file
.r	Ratfor source file
.f	Fortran source file
.s	Assembler source file
.y	Yacc-C source grammar
.yr	Yacc-Ratfor source grammar
.ye	Yacc-Efl source grammar
.l	Lex source grammar

The following diagram summarizes the default transformation paths. If there are two paths

connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.



If the file *x.o* were needed and there were an *x.c* in the description or directory, it would be compiled. If there were also an *x.l*, that grammar would be run through Lex before compiling the result. However, if there were no *x.c* but there were an *x.l*, *make* would discard the intermediate C-language file and use the direct link in the graph above.

It is possible to change the names of some of the compilers used in the default, or the flag arguments with which they are invoked by knowing the macro names used. The compiler names are the macros AS, CC, RC, EC, YACC, YACCR, YACCE, and LEX. The command

```
make CC=newcc
```

will cause the "newcc" command to be used instead of the usual C compiler. The macros CFLAGS, RFLAGS, EFLAGS, YFLAGS, and LFLAGS may be set to cause these commands to be issued with optional flags. Thus,

```
make "CFLAGS= -O"
```

causes the optimizing C compiler to be used.

Another special macro is 'VPATH'. The "VPATH" macro should be set to a list of directories separated by colons. When *make* searches for a file as a result of a dependency relation, it will first search the current directory and then each of the directories on the "VPATH" list. If the file is found, the actual path to the file will be used, rather than just the filename. If "VPATH" is not defined, then only the current directory is searched. Note that "VPATH" is intended to act like the System V "VPATH" support, but there is no guarantee that it functions identically.

One use for "VPATH" is when one has several programs that compile from the same source. The source can be kept in one directory and each set of object files (along with a separate would be in a separate subdirectory. The "VPATH" macro would point to the source directory in this case.

Example

As an example of the use of *make*, we will present the description file used to maintain the *make* command itself. The code for *make* is spread over a number of C source files and a Yacc grammar. The description file contains:

```
# Description file for the Make command
P = und -3 | opr -r2      # send to GCOS to be printed
FILES = Makefile version.c defs main.c doname.c misc.c files.c dosys.cgram.y lex.c gcos.c
OBJECTS = version.o main.o doname.o misc.o files.o dosys.o gram.o
LIBES= -lS
LINT = lint -p
CFLAGS = -O
make: $(OBJECTS)
	cc $(CFLAGS) $(OBJECTS) $(LIBES) -o make
	size make
$(OBJECTS): defs
gram.o: lex.c
cleanup:
	-rm *.o gram.c
	-du
install:
	@size make /usr/bin/make
	cp make /usr/bin/make ; rm make
print: $(FILES)# print recently changed files
	pr $? | $P
	touch print
test:
	make -dp | grep -v TIME >1zap
	/usr/bin/make -dp | grep -v TIME >2zap
	diff 1zap 2zap
	rm 1zap 2zap
lint : dosys.c doname.c files.c main.c misc.c version.c gram.c
	$(LINT) dosys.c doname.c files.c main.c misc.c version.c gram.c
	rm gram.c
arch:
	ar uv /sys/source/s2/make.a $(FILES)
```

Make usually prints out each command before issuing it. The following output results from typing the simple command

```
make
```

in a directory containing only the source and description file:

```
cc -c version.c
cc -c main.c
cc -c doname.c
cc -c misc.c
cc -c files.c
cc -c dosys.c
yacc gram.y
mv y.tab.c gram.c
cc -c gram.c
cc version.o main.o doname.o misc.o files.o dosys.o gram.o -lS -o make
13188+3348+3044 = 19580b = 046174b
```

Although none of the source files or grammars were mentioned by name in the description file, *make* found them using its suffix rules and issued the needed commands. The string of digits results from the "size make" command; the printing of the command line itself was suppressed by an @ sign. The

@ sign on the *size* command in the description file suppressed the printing of the command, so only the sizes are written.

The last few entries in the description file are useful maintenance sequences. The "print" entry prints only the files that have been changed since the last "make print" command. A zero-length file *print* is maintained to keep track of the time of the printing; the $? macro in the command line then picks up only the names of the files changed since *print* was touched. The printed output can be sent to a different printer or to a file by changing the definition of the *P* macro:

```
make print "P = opr -sp"
    or
make print "P= cat >zap"
```

Suggestions and Warnings

The most common difficulties arise from *make*'s specific meaning of dependency. If file *x.c* has a "#include "defs"" line, then the object file *x.o* depends on *defs*; the source file *x.c* does not. (If *defs* is changed, it is not necessary to do anything to the file *x.c*, while it is necessary to recreate *x.o*.)

To discover what *make* would do, the "–n" option is very useful. The command

```
make -n
```

orders *make* to print out the commands it would issue without actually taking the time to execute them. If a change to a file is absolutely certain to be benign (e.g., adding a new definition to an include file), the "–t" (touch) option can save a lot of time: instead of issuing a large number of superfluous recompilations, *make* updates the modification times on the affected file. Thus, the command

```
make -ts
```

("touch silently") causes the relevant files to appear up to date. Obvious care is necessary, since this mode of operation subverts the intention of *make* and destroys all memory of the previous relationships.

The debugging flag ("–d") causes *make* to print out a very detailed description of what it is doing, including the file times. The output is verbose, and recommended only as a last resort.

Acknowledgments

I would like to thank S. C. Johnson for suggesting this approach to program maintenance control. I would like to thank S. C. Johnson and H. Gajewska for being the prime guinea pigs during development of *make*.

References

1. S. C. Johnson, "Yacc — Yet Another Compiler-Compiler", Bell Laboratories Computing Science Technical Report #32, July 1978.
2. M. E. Lesk, "Lex — A Lexical Analyzer Generator", Computing Science Technical Report #39, October 1975.

Appendix. Suffixes and Transformation Rules

The *make* program itself does not know what file name suffixes are interesting or how to transform a file with one suffix into a file with another suffix. This information is stored in an internal table that has the form of a description file. If the "–r" flag is used, this table is not used.

The list of suffixes is actually the dependency list for the name ".SUFFIXES"; *make* looks for a file with any of the suffixes on the list. If such a file exists, and if there is a transformation rule for that combination, *make* acts as described earlier. The transformation rule names are the concatenation of the two suffixes. The name of the rule to transform a ".*r*" file to a ".*o*" file is thus ".*r.o*". If the rule is present and no explicit command sequence has been given in the user's description files, the command sequence for the rule ".r.o" is used. If a command is generated by using one of these suffixing rules, the macro $* is given the value of the stem (everything but the suffix) of the name of the file to be made, and the macro $< is the name of the dependent that caused the action.

The order of the suffix list is significant, since it is scanned from left to right, and the first name that is formed that has both a file and a rule associated with it is used. If new names are to be appended, the user can just add an entry for ".SUFFIXES" in his own description file; the dependents will be added to the usual list. A ".SUFFIXES" line without any dependents deletes the current list. (It is necessary to clear the current list if the order of names is to be changed).

The following is an excerpt from the default rules file:

```
.SUFFIXES : .o .c .e .r .f .y .yr .ye .l .s
YACC=yacc
YACCR=yacc -r
YACCE=yacc -e
YFLAGS=
LEX=lex
LFLAGS=
CC=cc
AS=as -
CFLAGS=
RC=ec
RFLAGS=
EC=ec
EFLAGS=
FFLAGS=
.c.o :
        $(CC) $(CFLAGS) -c $<
.e.o .r.o .f.o :
        $(EC) $(RFLAGS) $(EFLAGS) $(FFLAGS) -c $<
.s.o :
        $(AS) -o $@ $<
.y.o :
        $(YACC) $(YFLAGS) $<
        $(CC) $(CFLAGS) -c y.tab.c
        rm y.tab.c
        mv y.tab.o $@
.y.c :
        $(YACC) $(YFLAGS) $<
        mv y.tab.c $@
```

An Introduction to the Revision Control System

Walter F. Tichy

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

ABSTRACT

The Revision Control System (RCS) manages software libraries. It greatly increases programmer productivity by centralizing and cataloging changes to a software project. This document describes the benefits of using a source code control system. It then gives a tutorial introduction to the use of RCS.

Functions of RCS

The Revision Control System (RCS) manages multiple revisions of text files. RCS automates the storing, retrieval, logging, identification, and merging of revisions. RCS is useful for text that is revised frequently, for example programs, documentation, graphics, papers, form letters, etc. It greatly increases programmer productivity by providing the following functions.

1. RCS stores and retrieves multiple revisions of program and other text. Thus, one can maintain one or more releases while developing the next release, with a minimum of space overhead. Changes no longer destroy the original -- previous revisions remain accessible.
   a. Maintains each module as a tree of revisions.
   b. Project libraries can be organized centrally, decentralized, or any way you like.
   c. RCS works for any type of text: programs, documentation, memos, papers, graphics, VLSI layouts, form letters, etc.
2. RCS maintains a complete history of changes. Thus, one can find out what happened to a module easily and quickly, without having to compare source listings or having to track down colleagues.
   a. RCS performs automatic record keeping.
   b. RCS logs all changes automatically.
   c. RCS guarantees project continuity.
3. RCS manages multiple lines of development.
4. RCS can merge multiple lines of development. Thus, when several parallel lines of development must be consolidated into one line, the merging of changes is automatic.
5. RCS flags coding conflicts. If two or more lines of development modify the same section of code, RCS can alert programmers about overlapping changes.
6. RCS resolves access conflicts. When two or more programmers wish to modify the same revision, RCS alerts the programmers and makes sure that one change will not wipe out the other one.
7. RCS provides high-level retrieval functions. Revisions can be retrieved according to ranges of revision numbers, symbolic names, dates, authors, and states.
8. RCS provides release and configuration control. Revisions can be marked as released, stable, experimental, etc. Configurations of modules can be described simply and directly.

9. RCS performs automatic identification of modules with name, revision number, creation time, author, etc. Thus, it is always possible to determine which revisions of which modules make up a given configuration.
10. Provides high-level management visibility. Thus, it is easy to track the status of a software project.
    a. RCS provides a complete change history.
    b. RCS records who did what when to which revision of which module.
11. RCS is fully compatible with existing software development tools. RCS is unobtrusive -- its interface to the file system is such that all your existing software tools can be used as before.
12. RCS' basic user interface is extremely simple. The novice only needs to learn two commands. Its more sophisticated features have been tuned towards advanced software development environments and the experienced software professional.
13. RCS simplifies software distribution if customers also maintain sources with RCS. This technique assures proper identification of versions and configurations, and tracking of customer changes. Customer changes can be merged into distributed versions locally or by the development group.
14. RCS needs little extra space for the revisions (only the differences). If intermediate revisions are deleted, the corresponding differences are compressed into the shortest possible form.

Getting Started with RCS

Suppose you have a file f.c that you wish to put under control of RCS. Invoke the checkin command:

```
ci f.c
```

This command creates f.c,v, stores f.c into it as revision 1.1, and deletes f.c. It also asks you for a description. The description should be a synopsis of the contents of the file. All later checkin commands will ask you for a log entry, which should summarize the changes that you made.

Files ending in ,v are called RCS files ("v" stands for "versions"), the others are called working files. To get back the working file f.c in the previous example, use the checkout command:

```
co f.c
```

This command extracts the latest revision from f.c,v and writes it into f.c. You can now edit f.c and check it in back in by invoking:

```
ci f.c
```

Ci increments the revision number properly. If *ci* complains with the message

```
ci error: no lock set by <your login>
```

then your system administrator has decided to create all RCS files with the locking attribute set to "strict". With strict locking, you you must lock the revision during the previous checkout. Thus, your last checkout should have been

```
co -l f.c
```

Locking assures that you, and only you, can check in the next update, and avoids nasty problems if several people work on the same file. Of course, it is too late now to do the checkout with locking, because you probably modified f.c already, and a second checkout would overwrite your changes. Instead, invoke

```
rcs -l f.c
```

This command will lock the latest revision for you, unless somebody else got ahead of you already. If someone else has the lock you will have to negotiate your changes with them.

If your RCS file is private, i.e., if you are the only person who is going to deposit revisions into it, strict locking is not needed and you can turn it off. If strict locking is turned off, the owner off the RCS file need not have a lock for checkin; all others still do. Turning strict locking off and on is done with the commands:

rcs -U f.c and rcs -L f.c

You can set the locking to strict or non-strict on every RCS file.

If you do not want to clutter your working directory with RCS files, create a subdirectory called RCS in your working directory, and move all your RCS files there. RCS commands will look first into that directory to find needed files. All the commands discussed above will still work, without any change*.

To avoid the deletion of the working file during checkin (should you want to continue editing), invoke

ci -l f.c

This command checks in f.c as usual, but performs an additional checkout with locking. Thus, it saves you one checkout operation. There is also an option **-u** for *ci* that does a checkin followed by a checkout without locking. This is useful if you want to compile the file after the checkin. Both options also update the identification markers in your file (see below).

You can give *ci* the number you want assigned to a checked in revision. Assume all your revisions were numbered 1.1, 1.2, 1.3, etc., and you would like to start release 2. The command

ci -r2 f.c or ci -r2.1 f.c

assigns the number 2.1 to the new revision. From then on, *ci* will number the subsequent revisions with 2.2, 2.3, etc. The corresponding *co* commands

co -r2 f.c and co -r2.1 f.c

retrieve the latest revision numbered 2.x and the revision 2.1, respectively. *Co* without a revision number selects the latest revision on the "trunk", i.e., the highest revision with a number consisting of 2 fields. Numbers with more than 2 fields are needed for branches. For example, to start a branch at revision 1.3, invoke

ci -r1.3.1 f.c

This command starts a branch numbered 1 at revision 1.3, and assigns the number 1.3.1.1 to the new revision. For more information about branches, see *rcsfile*(5).

P S13 1

Automatic Identification

RCS can put special strings for identification into your source and object code. To obtain such identification, place the marker

$Header$

into your text, for instance inside a comment. RCS will replace this marker with a string of the form

$Header: filename revisionnumber date time author state $

You never need to touch this string, because RCS keeps it up to date automatically. To propagate the marker into your object code, simply put it into a literal character string. In C, this is done as follows:

static char rcsid[] = "$Header$";

The command *ident* extracts such markers from any file, even object code. Thus, *ident* helps you to

* Pairs of RCS and working files can really be specified in 3 ways: a) both are given, b) only the working file is given, c) only the RCS file is given. Both files may have arbitrary path prefixes; RCS commands pair them up intelligently.

find out which revisions of which modules were used in a given program.

You may also find it useful to put the marker

Log

into your text, inside a comment. This marker accumulates the log messages that are requested during checkin. Thus, you can maintain the complete history of your file directly inside it. There are several additional identification markers; see *co* (1) for details.

How to combine MAKE and RCS

If your RCS files are in the same directory as your working files, you can put a default rule into your makefile. Do not use a rule of the form .c,v.c, because such a rule keeps a copy of every working file checked out, even those you are not working on. Instead, use this:

```
.SUFFIXES: .c,v

.c,v.o:
        co -q $*.c
        cc $(CFLAGS) -c $*.c
        rm -f $*.c

prog:   f1.o f2.o .....
        cc f1.o f2.o ..... -o prog
```

This rule has the following effect. If a file f.c does not exist, and f.o is older than f.c,v, MAKE checks out f.c, compiles f.c into f.o, and then deletes f.c. From then on, MAKE will use f.o until you change f.c,v.

If f.c exists (presumably because you are working on it), the default rule .c.o takes precedence, and f.c is compiled into f.o, but not deleted.

If you keep your RCS file in the directory ./RCS, all this will not work and you have to write explicit checkout rules for every file, like

```
f1.c:   RCS/f1.c,v; co -q f1.c
```

Unfortunately, these rules do not have the property of removing unneeded .c-files.

Additional Information on RCS

If you want to know more about RCS, for example how to work with a tree of revisions and how to use symbolic revision numbers, read the following paper:

Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

Taking a look at the manual page *RCSFILE*(5) should also help to understand the revision tree permitted by RCS.

NAME

ci – check in RCS revisions

SYNOPSIS

ci [options] file ...

DESCRIPTION

Ci stores new revisions into RCS files. Each file name ending in ',v' is taken to be an RCS file, all others are assumed to be working files containing new revisions. *Ci* deposits the contents of each working file into the corresponding RCS file.

Pairs of RCS files and working files may be specified in 3 ways (see also the example section of *co* (1)).

1) Both the RCS file and the working file are given. The RCS file name is of the form *path1/workfile*,v and the working file name is of the form *path2/workfile*, where *path1/* and *path2/* are (possibly different or empty) paths and *workfile* is a file name.

2) Only the RCS file is given. Then the working file is assumed to be in the current directory and its name is derived from the name of the RCS file by removing *path1/* and the suffix ',v'.

3) Only the working file is given. Then the name of the RCS file is derived from the name of the working file by removing *path2/* and appending the suffix ',v'.

If the RCS file is omitted or specified without a path, then *ci* looks for the RCS file first in the directory ./RCS and then in the current directory.

For *ci* to work, the caller's login must be on the access list, except if the access list is empty or the caller is the superuser or the owner of the file. To append a new revision to an existing branch, the tip revision on that branch must be locked by the caller. Otherwise, only a new branch can be created. This restriction is not enforced for the owner of the file, unless locking is set to *strict* (see *rcs* (1)). A lock held by someone else may be broken with the *rcs* command.

Normally, *ci* checks whether the revision to be deposited is different from the preceding one. If it is not different, *ci* either aborts the deposit (if **-q** is given) or asks whether to abort (if **-q** is omitted). A deposit can be forced with the **-f** option.

For each revision deposited, *ci* prompts for a log message. The log message should summarize the change and must be terminated with a line containing a single '.' or a control-D. If several files are checked in, *ci* asks whether to reuse the previous log message. If the std. input is not a terminal, *ci* suppresses the prompt and uses the same log message for all files. See also **-m**.

The number of the deposited revision can be given by any of the options **-r**, **-f**, **-k**, **-l**, **-u**, or **-q** (see **-r**).

If the RCS file does not exist, *ci* creates it and deposits the contents of the working file as the initial revision (default number: 1.1). The access list is initialized to empty. Instead of the log message, *ci* requests descriptive text (see **-t** below).

-r[*rev*] assigns the revision number *rev* to the checked-in revision, releases the corresponding lock, and deletes the working file. This is also the default.

If *rev* is omitted, *ci* derives the new revision number from the caller's last lock. If the caller has locked the tip revision of a branch, the new revision is appended to that branch. The new revision number is obtained by incrementing the tip revision number. If the caller locked a non-tip revision, a new branch is started at that revision by incrementing the highest branch number at that revision. The default initial branch and level numbers are 1. If the caller holds no lock, but he is the owner of the file and locking is not set to *strict*, then the revision is appended to the

trunk.

If *rev* indicates a revision number, it must be higher than the latest one on the branch to which *rev* belongs, or must start a new branch.

If *rev* indicates a branch instead of a revision, the new revision is appended to that branch. The level number is obtained by incrementing the tip revision number of that branch. If *rev* indicates a non-existing branch, that branch is created with the initial revision numbered *rev.1*.

Exception: On the trunk, revisions can be appended to the end, but not inserted.

-f[*rev*] forces a deposit; the new revision is deposited even it is not different from the preceding one.

-k[*rev*] searches the working file for keyword values to determine its revision number, creation date, author, and state (see *co* (1)), and assigns these values to the deposited revision, rather than computing them locally. A revision number given by a command option overrides the number in the working file. This option is useful for software distribution. A revision that is sent to several sites should be checked in with the **-k** option at these sites to preserve its original number, date, author, and state.

-l[*rev*] works like **-r**, except it performs an additional *co -l* for the deposited revision. Thus, the deposited revision is immediately checked out again and locked. This is useful for saving a revision although one wants to continue editing it after the checkin.

-u[*rev*] works like **-l**, except that the deposited revision is not locked. This is useful if one wants to process (e.g., compile) the revision immediately after checkin.

-q[*rev*] quiet mode; diagnostic output is not printed. A revision that is not different from the preceding one is not deposited, unless **-f** is given.

-m*msg* uses the string *msg* as the log message for all revisions checked in.

-n*name* assigns the symbolic name *name* to the number of the checked-in revision. *Ci* prints an error message if *name* is already assigned to another number.

-N*name* same as **-n**, except that it overrides a previous assignment of *name*.

-s*state* sets the state of the checked-in revision to the identifier *state*. The default is *Exp*.

-t[*txtfile*] writes descriptive text into the RCS file (deletes the existing text). If *txtfile* is omitted, *ci* prompts the user for text supplied from the std. input, terminated with a line containing a single '.' or control-D. Otherwise, the descriptive text is copied from the file *txtfile*. During initialization, descriptive text is requested even if **-t** is not given. The prompt is suppressed if std. input is not a terminal.

DIAGNOSTICS

For each revision, *ci* prints the RCS file, the working file, and the number of both the deposited and the preceding revision. The exit status always refers to the last file checked in, and is 0 if the operation was successful, 1 otherwise.

FILE MODES

An RCS file created by *ci* inherits the read and execute permissions from the working file. If the RCS file exists already, *ci* preserves its read and execute permissions. *Ci* always turns off all write permissions of RCS files.

FILES

The caller of the command must have read/write permission for the directories containing the RCS file and the working file, and read permission for the RCS file itself. A number of temporary files are created. A semaphore file is created in the directory containing the RCS file. *Ci* always creates a new RCS file and unlinks the old one. This strategy makes links to RCS files useless.

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.1 ; Release Date: 83/04/04 .

SEE ALSO

co (1), ident(1), rcs (1), rcsdiff (1), rcsintro (1), rcsmerge (1), rlog (1), rcsfile (5), sccstorcs (8).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

BUGS

NAME

co – check out RCS revisions

SYNOPSIS

co [options] file ...

DESCRIPTION

Co retrieves revisions from RCS files. Each file name ending in ',v' is taken to be an RCS file. All other files are assumed to be working files. *Co* retrieves a revision from each RCS file and stores it into the corresponding working file.

Pairs of RCS files and working files may be specified in 3 ways (see also the example section).

1) Both the RCS file and the working file are given. The RCS file name is of the form *path1/workfile*,v and the working file name is of the form *path2/workfile*, where *path1/* and *path2/* are (possibly different or empty) paths and *workfile* is a file name.

2) Only the RCS file is given. Then the working file is created in the current directory and its name is derived from the name of the RCS file by removing *path1/* and the suffix ',v'.

3) Only the working file is given. Then the name of the RCS file is derived from the name of the working file by removing *path2/* and appending the suffix ',v'.

If the RCS file is omitted or specified without a path, then *co* looks for the RCS file first in the directory ./RCS and then in the current directory.

Revisions of an RCS file may be checked out locked or unlocked. Locking a revision prevents overlapping updates. A revision checked out for reading or processing (e.g., compiling) need not be locked. A revision checked out for editing and later checkin must normally be locked. Locking a revision currently locked by another user fails. (A lock may be broken with the *rcs* (1) command.) *Co* with locking requires the caller to be on the access list of the RCS file, unless he is the owner of the file or the superuser, or the access list is empty. *Co* without locking is not subject to accesslist restrictions.

A revision is selected by number, checkin date/time, author, or state. If none of these options are specified, the latest revision on the trunk is retrieved. When the options are applied in combination, the latest revision that satisfies all of them is retrieved. The options for date/time, author, and state retrieve a revision on the *selected branch*. The selected branch is either derived from the revision number (if given), or is the highest branch on the trunk. A revision number may be attached to one of the options **-l**, **-p**, **-q**, or **-r**.

A *co* command applied to an RCS file with no revisions creates a zero-length file. *Co* always performs keyword substitution (see below).

-l[*rev*] locks the checked out revision for the caller. If omitted, the checked out revision is not locked. See option **-r** for handling of the revision number *rev*.

-p[*rev*] prints the retrieved revision on the std. output rather than storing it in the working file. This option is useful when *co* is part of a pipe.

-q[*rev*] quiet mode; diagnostics are not printed.

-d*date* retrieves the latest revision on the selected branch whose checkin date/time is less than or equal to *date*. The date and time may be given in free format and are converted to local time. Examples of formats for *date*:

22-April-1982, 17:20-CDT,
2:25 AM, Dec. 29, 1983,
Tue-PDT, 1981, 4pm Jul 21 (free format),
Fri, April 16 15:52:25 EST 1982 (output of ctime).

Most fields in the date and time may be defaulted. *Co* determines the defaults in the

order year, month, day, hour, minute, and second (most to least significant). At least one of these fields must be provided. For omitted fields that are of higher significance than the highest provided field, the current values are assumed. For all other omitted fields, the lowest possible values are assumed. For example, the date "20, 10:30" defaults to 10:30:00 of the 20th of the current month and current year. The date/time must be quoted if it contains spaces.

-r[*rev*] retrieves the latest revision whose number is less than or equal to *rev*. If *rev* indicates a branch rather than a revision, the latest revision on that branch is retrieved. *Rev* is composed of one or more numeric or symbolic fields separated by '.'. The numeric equivalent of a symbolic field is specified with the **-n** option of the commands *ci* and *rcs*.

-s*state* retrieves the latest revision on the selected branch whose state is set to *state*.

-w[*login*] retrieves the latest revision on the selected branch which was checked in by the user with login name *login*. If the argument *login* is omitted, the caller's login is assumed.

-j*joinlist* generates a new revision which is the join of the revisions on *joinlist*. *Joinlist* is a comma-separated list of pairs of the form *rev2:rev3*, where *rev2* and *rev3* are (symbolic or numeric) revision numbers. For the initial such pair, *rev1* denotes the revision selected by the options **-l**, ..., **-w**. For all other pairs, *rev1* denotes the revision generated by the previous pair. (Thus, the output of one join becomes the input to the next.)

For each pair, *co* joins revisions *rev1* and *rev3* with respect to *rev2*. This means that all changes that transform *rev2* into *rev1* are applied to a copy of *rev3*. This is particularly useful if *rev1* and *rev3* are the ends of two branches that have *rev2* as a common ancestor. If *rev1* < *rev2* < *rev3* on the same branch, joining generates a new revision which is like *rev3*, but with all changes that lead from *rev1* to *rev2* undone. If changes from *rev2* to *rev1* overlap with changes from *rev2* to *rev3*, *co* prints a warning and includes the overlapping sections, delimited by the lines <<<<<<< *rev1*, =======, and >>>>>>> *rev3*.

For the initial pair, *rev2* may be omitted. The default is the common ancestor. If any of the arguments indicate branches, the latest revisions on those branches are assumed. If the option **-l** is present, the initial *rev1* is locked.

P S13 1

KEYWORD SUBSTITUTION

Strings of the form *$keyword$* and *$keyword:...$* embedded in the text are replaced with strings of the form *$keyword: value $*, where *keyword* and *value* are pairs listed below. Keywords may be embedded in literal strings or comments to identify a revision.

Initially, the user enters strings of the form *$keyword$*. On checkout, *co* replaces these strings with strings of the form *$keyword: value $*. If a revision containing strings of the latter form is checked back in, the value fields will be replaced during the next checkout. Thus, the keyword values are automatically updated on checkout.

Keywords and their corresponding values:

$Author$ The login name of the user who checked in the revision.

$Date$ The date and time the revision was checked in.

$Header$ A standard header containing the RCS file name, the revision number, the date, the author, and the state.

$Locker$ The login name of the user who locked the revision (empty if not locked).

Log	The log message supplied during checkin, preceded by a header containing the RCS file name, the revision number, the author, and the date. Existing log messages are NOT replaced. Instead, the new log message is inserted after *$Log:...$*. This is useful for accumulating a complete change log in a source file.
$Revision$	The revision number assigned to the revision.
$Source$	The full pathname of the RCS file.
$State$	The state assigned to the revision with *rcs -s* or *ci -s*.

DIAGNOSTICS

The RCS file name, the working file name, and the revision number retrieved are written to the diagnostic output. The exit status always refers to the last file checked out, and is 0 if the operation was successful, 1 otherwise.

EXAMPLES

Suppose the current directory contains a subdirectory 'RCS' with an RCS file 'io.c,v'. Then all of the following commands retrieve the latest revision from 'RCS/io.c,v' and store it into 'io.c'.

```
co io.c;   co RCS/io.c,v;   co io.c,v;
co io.c RCS/io.c,v;   co io.c io.c,v;
co RCS/io.c,v io.c;   co io.c,v io.c;
```

FILE MODES

The working file inherits the read and execute permissions from the RCS file. In addition, the owner write permission is turned on, unless the file is checked out unlocked and locking is set to *strict* (see *rcs* (1)).

If a file with the name of the working file exists already and has write permission, *co* aborts the checkout if **-q** is given, or asks whether to abort if **-q** is not given. If the existing working file is not writable, it is deleted before the checkout.

P S13 1

FILES

The caller of the command must have write permission in the working directory, read permission for the RCS file, and either read permission (for reading) or read/write permission (for locking) in the directory which contains the RCS file.

A number of temporary files are created. A semaphore file is created in the directory of the RCS file to prevent simultaneous update.

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.1 ; Release Date: 83/04/04 .

SEE ALSO

ci (1), ident(1), rcs (1), rcsdiff (1), rcsintro (1), rcsmerge (1), rlog (1), rcsfile (5), sccstorcs (8).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

LIMITATIONS

The option **-d** gets confused in some circumstances, and accepts no date before 1970. There is no way to suppress the expansion of keywords, except by writing them differently. In nroff and troff, this is done by embedding the null-character '\&' into the keyword.

BUGS

The option **-j** does not work for files that contain lines with a single '.'.

NAME

ident – identify files

SYNOPSIS

ident file ...

DESCRIPTION

Ident searches the named files for all occurrences of the pattern *$keyword:...$*, where *keyword* is one of

Author
Date
Header
Locker
Log
Revision
Source
State

These patterns are normally inserted automatically by the RCS command *co (1)*, but can also be inserted manually.

Ident works on text files as well as object files. For example, if the C program in file f.c contains

```
char rcsid[] = "$Header: Header information $";
```

and f.c is compiled into f.o, then the command

```
ident f.c f.o
```

will print

```
f.c:
     $Header: Header information $
f.o:
     $Header: Header information $
```

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.0 ; Release Date: 82/12/04 .

SEE ALSO

ci (1), co (1), rcs (1), rcsdiff(1), rcsintro (1), rcsmerge (1), rlog (1), rcsfile (5).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

BUGS

NAME

merge – three-way file merge

SYNOPSIS

merge [**-p**] file1 file2 file3

DESCRIPTION

Merge incorporates all changes that lead from *file2* to *file3* into *file1*. The result goes to std. output if **-p** is present, into *file1* otherwise. *Merge* is useful for combining separate changes to an original. Suppose *file2* is the original, and both *file1* and *file3* are modifications of *file2*. Then *merge* combines both changes.

An overlap occurs if both *file1* and *file3* have changes in a common segment of lines. *Merge* prints how many overlaps occurred, and includes both alternatives in the result. The alternatives are delimited as follows:

```
<<<<<<< file1
lines in file1
=======
lines in file3
>>>>>>> file3
```

If there are overlaps, the user should edit the result and delete one of the alternatives.

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.0 ; Release Date: 82/11/25 .

SEE ALSO

diff3 (1), diff (1), rcsmerge (1), co (1).

NAME

rcs – change RCS file attributes

SYNOPSIS

rcs [options] file ...

DESCRIPTION

Rcs creates new RCS files or changes attributes of existing ones. An RCS file contains multiple revisions of text, an access list, a change log, descriptive text, and some control attributes. For *rcs* to work, the caller's login name must be on the access list, except if the access list is empty, the caller is the owner of the file or the superuser, or the **-i** option is present.

Files ending in ',v' are RCS files, all others are working files. If a working file is given, *rcs* tries to find the corresponding RCS file first in directory ./RCS and then in the current directory, as explained in *co* (1).

-i creates and initializes a new RCS file, but does not deposit any revision. If the RCS file has no path prefix, *rcs* tries to place it first into the subdirectory ./RCS, and then into the current directory. If the RCS file already exists, an error message is printed.

-a*logins* appends the login names appearing in the comma-separated list *logins* to the access list of the RCS file.

-A*oldfile* appends the access list of *oldfile* to the access list of the RCS file.

-e[*logins*] erases the login names appearing in the comma-separated list *logins* from the access list of the RCS file. If *logins* is omitted, the entire access list is erased.

-c*string* sets the comment leader to *string*. The comment leader is printed before every log message line generated by the keyword Log during checkout (see *co*). This is useful for programming languages without multi-line comments. During *rcs -i* or initial *ci*, the comment leader is guessed from the suffix of the working file.

-l[*rev*] locks the revision with number *rev*. If a branch is given, the latest revision on that branch is locked. If *rev* is omitted, the latest revision on the trunk is locked. Locking prevents overlapping changes. A lock is removed with *ci* or *rcs -u* (see below).

-u[*rev*] unlocks the revision with number *rev*. If a branch is given, the latest revision on that branch is unlocked. If *rev* is omitted, the latest lock held by the caller is removed. Normally, only the locker of a revision may unlock it. Somebody else unlocking a revision breaks the lock. This causes a mail message to be sent to the original locker. The message contains a commentary solicited from the breaker. The commentary is terminated with a line containing a single '.' or control-D.

-L sets locking to *strict*. Strict locking means that the owner of an RCS file is not exempt from locking for checkin. This option should be used for files that are shared.

-U sets locking to non-strict. Non-strict locking means that the owner of a file need not lock a revision for checkin. This option should NOT be used for files that are shared. The default (**-L** or **-U**) is determined by your system administrator.

-n*name*[:*rev*]
associates the symbolic name *name* with the branch or revision *rev*. *Rcs* prints an error message if *name* is already associated with another number. If *rev* is omitted, the symbolic name is deleted.

-N*name*[:*rev*]
same as **-n**, except that it overrides a previous assignment of *name*.

-o*range* deletes ("outdates") the revisions given by *range*. A range consisting of a single revision number means that revision. A range consisting of a branch number means the latest revision on that branch. A range of the form *rev1-rev2* means revisions *rev1* to *rev2* on the same branch, *-rev* means from the beginning of the branch containing *rev* up to and including *rev*, and *rev-* means from revision *rev* to the end of the branch containing *rev*. None of the outdated revisions may have branches or locks.

-q quiet mode; diagnostics are not printed.

-s*state*[:*rev*]
sets the state attribute of the revision *rev* to *state*. If *rev* is omitted, the latest revision on the trunk is assumed; If *rev* is a branch number, the latest revision on that branch is assumed. Any identifier is acceptable for *state*. A useful set of states is *Exp* (for experimental), *Stab* (for stable), and *Rel* (for released). By default, *ci* sets the state of a revision to *Exp*.

-t[*txtfile*] writes descriptive text into the RCS file (deletes the existing text). If *txtfile* is omitted, *rcs* prompts the user for text supplied from the std. input, terminated with a line containing a single '.' or control-D. Otherwise, the descriptive text is copied from the file *txtfile*. If the -i option is present, descriptive text is requested even if -t is not given. The prompt is suppressed if the std. input is not a terminal.

DIAGNOSTICS

The RCS file name and the revisions outdated are written to the diagnostic output. The exit status always refers to the last RCS file operated upon, and is 0 if the operation was successful, 1 otherwise.

FILES

The caller of the command must have read/write permission for the directory containing the RCS file and read permission for the RCS file itself. *Rcs* creates a semaphore file in the same directory as the RCS file to prevent simultaneous update. For changes, *rcs* always creates a new file. On successful completion, *rcs* deletes the old one and renames the new one. This strategy makes links to RCS files useless.

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.1 ; Release Date: 83/04/04 .

SEE ALSO

co (1), ci (1), ident(1), rcsdiff (1), rcsintro (1), rcsmerge (1), rlog (1), rcsfile (5), sccstorcs (8).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

NAME

rcsdiff – compare RCS revisions

SYNOPSIS

rcsdiff [-b] [-cefhn] [-r*rev1*] [-r*rev2*] file ...

DESCRIPTION

Rcsdiff runs *diff* (1) to compare two revisions of each RCS file given. A file name ending in ',v' is an RCS file name, otherwise a working file name. *Rcsdiff* derives the working file name from the RCS file name and vice versa, as explained in *co* (1). Pairs consisting of both an RCS and a working file name may also be specified.

The options **-b**, **-c**, **-e**, **-f**, and **-h** have the same effect as described in *diff* (1); option **-n** generates an edit script of the format used by RCS.

If both *rev1* and *rev2* are omitted, *rcsdiff* compares the latest revision on the trunk with the contents of the corresponding working file. This is useful for determining what you changed since the last checkin.

If *rev1* is given, but *rev2* is omitted, *rcsdiff* compares revision *rev1* of the RCS file with the contents of the corresponding working file.

If both *rev1* and *rev2* are given, *rcsdiff* compares revisions *rev1* and *rev2* of the RCS file.

Both *rev1* and *rev2* may be given numerically or symbolically.

EXAMPLES

The command

```
rcsdiff f.c
```

runs *diff* on the latest trunk revision of RCS file f.c,v and the contents of working file f.c.

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.0 ; Release Date: 83/01/15 .

SEE ALSO

ci (1), co (1), diff (1), ident (1), rcs (1), rcsintro (1), rcsmerge (1), rlog (1), rcsfile (5).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

NAME

rcsfile – format of RCS file

DESCRIPTION

An RCS file is an ASCII file. Its contents is described by the grammar below. The text is free format, i.e., spaces, tabs and new lines have no significance except in strings. Strings are enclosed by '@'. If a string contains a '@', it must be doubled.

The meta syntax uses the following conventions: '|' (bar) separates alternatives; '{' and '}' enclose optinal phrases; '{' and '}*' enclose phrases that may be repeated zero or more times; '{' and '}+' enclose phrases that must appear at least once and may be repeated; '<' and '>' enclose nonterminals.

```
<rcstext>     ::=  <admin> {<delta>}* <desc> {<deltatext>}*

<admin>       ::=  head          {<num>};
                   access        {<id>}*;
                   symbols       {<id> : <num>}*;
                   locks         {<id> : <num>}*;
                   comment       {<string>};

<delta>       ::=  <num>
                   date          <num>;
                   author        <id>;
                   state         {<id>};
                   branches      {<num>}*;
                   next          {<num>};

<desc>        ::=  desc          <string>

<deltatext>   ::=  <num>
                   log           <string>
                   text          <string>


<num>         ::=  {<digit>{.}}+

<digit>       ::=  0 | 1 | ... | 9

<id>          ::=  <letter>{<idchar>}*

<letter>      ::=  A | B | ... | Z | a | b | ... | z

<idchar>      ::=  Any printing ASCII character except space,
                   tab, carriage return, new line, and <special>.

<special>     ::=  ; | : | , | @

<string>      ::=  @{any ASCII character, with '@' doubled}*@
```

Identifiers are case sensitive. Keywords are in lower case only. The sets of keywords and identifiers may overlap.

The <delta> nodes form a tree. All nodes whose numbers consist of a single pair (e.g., 2.3, 2.1, 1.3, etc.) are on the "trunk", and are linked through the "next" field in order of decreasing numbers. The "head" field in the <admin> node points to the head of that sequence (i.e., contains the highest pair).

All <delta> nodes whose numbers consist of 2n fields (n≥2) (e.g., 3.1.1.1, 2.1.2.2, etc.) are linked as follows. All nodes whose first (2n)-1 number fields are identical are linked through the "next" field in order of increasing numbers. For each such sequence, the <delta> node whose number is identical to the first 2(n-1) number fields of the deltas on that sequence is called the branchpoint. The "branches" field of a node contains a list of the numbers of the first nodes of all sequences for which it is a branchpoint. This list is ordered in increasing numbers.

Example:

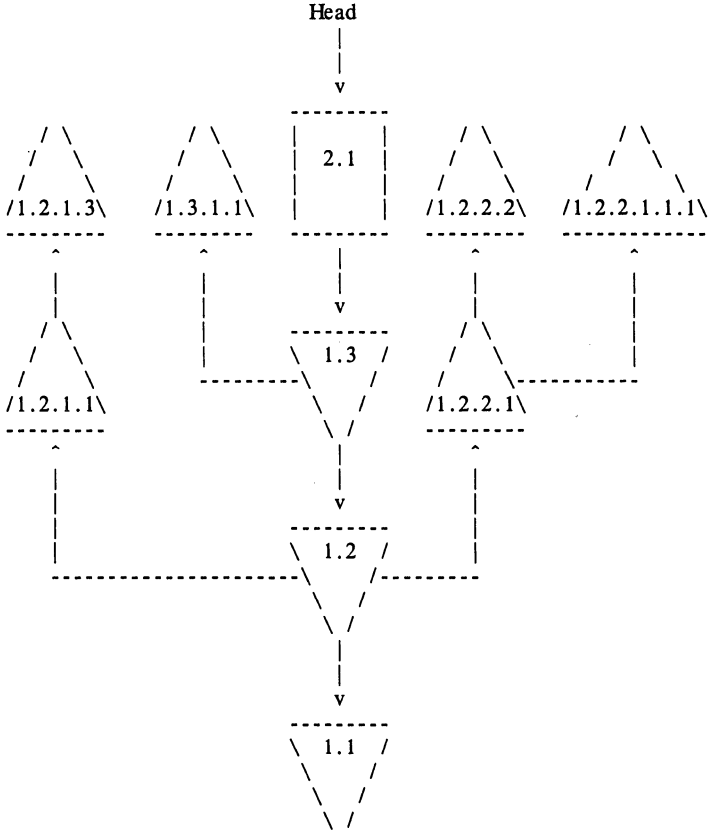


Fig. 1: A revision tree

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.0 ; Release Date: 82/11/18 .

SEE ALSO

ci (1), co (1), ident (1), rcs (1), rcsdiff (1), rcsintro (1), rcsmerge (1), rlog (1), sccstorcs (8).

NAME

rcsmerge – merge RCS revisions

SYNOPSIS

rcsmerge -r*rev1* [**-r***rev2*] [**-p**] file

DESCRIPTION

Rcsmerge incorporates the changes between *rev1* and *rev2* of an RCS file into the corresponding working file. If **-p** is given, the result is printed on the std. output, otherwise the result overwrites the working file.

A file name ending in ',v' is an RCS file name, otherwise a working file name. *Merge* derives the working file name from the RCS file name and vice versa, as explained in *co* (1). A pair consisting of both an RCS and a working file name may also be specified.

Rev1 may not be omitted. If *rev2* is omitted, the latest revision on the trunk is assumed. Both *rev1* and *rev2* may be given numerically or symbolically.

Rcsmerge prints a warning if there are overlaps, and delimits the overlapping regions as explained in *co -j*. The command is useful for incorporating changes into a checked-out revision.

EXAMPLES

Suppose you have released revision 2.8 of f.c. Assume furthermore that you just completed revision 3.4, when you receive updates to release 2.8 from someone else. To combine the updates to 2.8 and your changes between 2.8 and 3.4, put the updates to 2.8 into file f.c and execute

```
rcsmerge -p -r2.8 -r3.4 f.c >f.merged.c
```

Then examine f.merged.c. Alternatively, if you want to save the updates to 2.8 in the RCS file, check them in as revision 2.8.1.1 and execute *co -j*:

```
ci -r2.8.1.1 f.c
co -r3.4 -j2.8:2.8.1.1 f.c
```

As another example, the following command undoes the changes between revision 2.4 and 2.8 in your currently checked out revision in f.c.

```
rcsmerge -r2.8 -r2.4 f.c
```

Note the order of the arguments, and that f.c will be overwritten.

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.0 ; Release Date: 83/01/15 .

SEE ALSO

ci (1), co (1), merge (1), ident (1), rcs (1), rcsdiff (1), rlog (1), rcsfile (5).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

BUGS

Rcsmerge does not work for files that contain lines with a single '.'.

NAME

rlog – print log messages and other information about RCS files

SYNOPSIS

rlog [options] file ...

DESCRIPTION

Rlog prints information about RCS files. Files ending in ',v' are RCS files, all others are working files. If a working file is given, *rlog* tries to find the corresponding RCS file first in directory ./RCS and then in the current directory, as explained in *co* (1).

Rlog prints the following information for each RCS file: RCS file name, working file name, head (i.e., the number of the latest revision on the trunk), access list, locks, symbolic names, suffix, total number of revisions, number of revisions selected for printing, and descriptive text. This is followed by entries for the selected revisions in reverse chronological order for each branch. For each revision, *rlog* prints revision number, author, date/time, state, number of lines added/deleted (with respect to the previous revision), locker of the revision (if any), and log message. Without options, *rlog* prints complete information. The options below restrict this output.

-L ignores RCS files that have no locks set; convenient in combination with **-R**, **-h**, or **-l**.

-R only prints the name of the RCS file; convenient for translating a working file name into an RCS file name.

-h prints only RCS file name, working file name, head, access list, locks, symbolic names, and suffix.

-t prints the same as **-h**, plus the descriptive text.

-d*dates* prints information about revisions with a checkin date/time in the ranges given by the semicolon-separated list of *dates*. A range of the form *d1*<*d2* or *d2*>*d1* selects the revisions that were deposited between *d1* and *d2*, (inclusive). A range of the form <*d* or *d*> selects all revisions dated *d* or earlier. A range of the form *d*< or >*d* selects all revisions dated *d* or later. A range of the form *d* selects the single, latest revision dated *d* or earlier. The date/time strings *d*, *d1*, and *d2* are in the free format explained in *co* (1). Quoting is normally necessary, especially for < and >. Note that the separator is a semicolon.

-l[*lockers*] prints information about locked revisions. If the comma-separated list *lockers* of login names is given, only the revisions locked by the given login names are printed. If the list is omitted, all locked revisions are printed.

-r*revisions*

prints information about revisions given in the comma-separated list *revisions* of revisions and ranges. A range *rev1*-*rev2* means revisions *rev1* to *rev2* on the same branch, -*rev* means revisions from the beginning of the branch up to and including *rev*, and *rev*- means revisions starting with *rev* to the end of the branch containing *rev*. An argument that is a branch means all revisions on that branch. A range of branches means all revisions on the branches in that range.

-s*states* prints information about revisions whose state attributes match one of the states given in the comma-separated list *states*.

-w[*logins*] prints information about revisions checked in by users with login names appearing in the comma-separated list *logins*. If *logins* is omitted, the user's login is assumed.

Rlog prints the intersection of the revisions selected with the options **-d**, **-l**, **-s**, **-w**, intersected with the union of the revisions selected by **-b** and **-r**.

EXAMPLES

```
rlog -L -R RCS/*,v
rlog -L -h RCS/*,v
rlog -L -l RCS/*,v
rlog RCS/*,v
```

The first command prints the names of all RCS files in the subdirectory 'RCS' which have locks. The second command prints the headers of those files, and the third prints the headers plus the log messages of the locked revisions. The last command prints complete information.

DIAGNOSTICS

The exit status always refers to the last RCS file operated upon, and is 0 if the operation was successful, 1 otherwise.

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.2 ; Release Date: 83/05/11 .

SEE ALSO

ci (1), co (1), ident(1), rcs (1), rcsdiff (1), rcsintro (1), rcsmerge (1), rcsfile (5), sccstorcs (8).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

NAME

sccstorcs – build RCS file from SCCS file

SYNOPSIS

sccstorcs [–t] [–v] *s.file ...*

DESCRIPTION

Sccstorcs builds an RCS file from each SCCS file argument. The deltas and comments for each delta are preserved and installed into the new RCS file in order. Also preserved are the user access list and descriptive text, if any, from the SCCS file.

The following flags are meaningful:

–t Trace only. Prints detailed information about the SCCS file and lists the commands that would be executed to produce the RCS file. No commands are actually executed and no RCS file is made.

–v Verbose. Prints each command that is run while it is building the RCS file.

FILES

For each s.*somefile*, *Sccstorcs* writes the files *somefile* and *somefile*,v which should not already exist. *Sccstorcs* will abort, rather than overwrite those files if they do exist.

SEE ALSO

ci (1), co (1), rcs (1).
Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

DIAGNOSTICS

All diagnostics are written to stderr. Non-zero exit status on error.

BUGS

Sccstorcs does not preserve all SCCS options specified in the SCCS file. Most notably, it does not preserve removed deltas, MR numbers, and cutoff points.

AUTHOR

Ken Greer

An Introduction to the Source Code Control System

Eric Allman
Project Ingres
University of California at Berkeley

This document gives a quick introduction to using the Source Code Control System (SCCS). The presentation is geared to programmers who are more concerned with what to do to get a task done rather than how it works; for this reason some of the examples are not well explained. For details of what the magic options do, see the section on "Further Information".

> This is a working document. Please send any comments or suggestions to eric@Berkeley.Edu.

1. Introduction

SCCS is a source management system. Such a system maintains a record of versions of a system; a record is kept with each set of changes of what the changes are, why they were made, and who made them and when. Old versions can be recovered, and different versions can be maintained simultaneously. In projects with more than one person, SCCS will insure that two people are not editing the same file at the same time.

All versions of your program, plus the log and other information, is kept in a file called the "s-file". There are three major operations that can be performed on the s-file:

(1) Get a file for compilation (not for editing). This operation retrieves a version of the file from the s-file. By default, the latest version is retrieved. This file is intended for compilation, printing, or whatever; it is specifically NOT intended to be edited or changed in any way; any changes made to a file retrieved in this way will probably be lost.

(2) Get a file for editing. This operation also retrieves a version of the file from the s-file, but this file is intended to be edited and then incorporated back into the s-file. Only one person may be editing a file at one time.

(3) Merge a file back into the s-file. This is the companion operation to (2). A new version number is assigned, and comments are saved explaining why this change was made.

2. Learning the Lingo

There are a number of terms that are worth learning before we go any farther.

This is version 1.21 of this document. It was last modified on 12/5/80.

2.1. S-file

The s-file is a single file that holds all the different versions of your file. The s-file is stored in differential format; *i.e.*, only the differences between versions are stored, rather than the entire text of the new version. This saves disk space and allows selective changes to be removed later. Also included in the s-file is some header information for each version, including the comments given by the person who created the version explaining why the changes were made.

2.2. Deltas

Each set of changes to the s-file (which is approximately [but not exactly!] equivalent to a version of the file) is called a *delta*. Although technically a delta only includes the *changes* made, in practice it is usual for each delta to be made with respect to all the deltas that have occurred before[1]. However, it is possible to get a version of the file that has selected deltas removed out of the middle of the list of changes — equivalent to removing your changes later.

2.3. SID's (or, version numbers)

A SID (SCCS Id) is a number that represents a delta. This is normally a two-part number consisting of a "release" number and a "level" number. Normally the release number stays the same, however, it is possible to move into a new release if some major change is being made.

Since all past deltas are normally applied, the SID of the final delta applied can be used to represent a version number of the file as a whole.

2.4. Id keywords

When you get a version of a file with intent to compile and install it (*i.e.*, something other than edit it), some special keywords are expanded inline by SCCS. These *Id Keywords* can be used to include the current version number or other information into the file. All id keywords are of the form **%*x*%**, where *x* is an upper case letter. For example, **%I%** is the SID of the latest delta applied, **%W%** includes the module name, SID, and a mark that makes it findable by a program, and **%G%** is the date of the latest delta applied. There are many others, most of which are of dubious usefulness.

When you get a file for editing, the id keywords are not expanded; this is so that after you put them back in to the s-file, they will be expanded automatically on each new version. But notice: if you were to get them expanded accidently, then your file would appear to be the same version forever more, which would of course defeat the purpose. Also, if you should install a version of the program without expanding the id keywords, it will be impossible to tell what version it is (since all it will have is "%W%" or whatever).

P S14 1

3. Creating SCCS Files

To put source files into SCCS format, run the following shell script from csh:

```
mkdir SCCS save
foreach i (*.[ch])
        sccs admin -i$i $i
        mv $i save/$i
end
```

[1]This matches normal usage, where the previous changes are not saved at all, so all changes are automatically based on all other changes that have happened through history.

This will put the named files into s-files in the subdirectory "SCCS" The files will be removed from the current directory and hidden away in the directory "save", so the next thing you will probably want to do is to get all the files (described below). When you are convinced that SCCS has correctly created the s-files, you should remove the directory "save".

If you want to have id keywords in the files, it is best to put them in before you create the s-files. If you do not, *admin* will print "No Id Keywords (cm7)", which is a warning message only.

4. Getting Files for Compilation

To get a copy of the latest version of a file, run

```
sccs get prog.c
```

SCCS will respond:

```
1.1
87 lines
```

meaning that version 1.1 was retrieved[2] and that it has 87 lines. The file *prog.c* will be created in the current directory. The file will be read-only to remind you that you are not supposed to change it.

This copy of the file should not be changed, since SCCS is unable to merge the changes back into the s-file. If you do make changes, they will be lost the next time someone does a *get*.

5. Changing Files (or, Creating Deltas)

5.1. Getting a copy to edit

To edit a source file, you must first get it, requesting permission to edit it[3]:

```
sccs edit prog.c
```

The response will be the same as with *get* except that it will also say:

```
New delta 1.2
```

You then edit it, using a standard text editor:

```
vi prog.c
```

5.2. Merging the changes back into the s-file

When the desired changes are made, you can put your changes into the SCCS file using the *delta* command:

```
sccs delta prog.c
```

Delta will prompt you for "comments?" before it merges the changes in. At this prompt you should type a one-line description of what the changes mean (more lines can be entered

PS14 1

---

[2]Actually, the SID of the final delta applied was 1.1.

[3]The "edit" command is equivalent to using the –e flag to *get*, as:

```
sccs get -e prog.c
```

Keep this in mind when reading other documentation.

by ending each line except the last with a backslash[4]). *Delta* will then type:

```
1.2
5 inserted
3 deleted
84 unchanged
```

saying that delta 1.2 was created, and it inserted five lines, removed three lines, and left 84 lines unchanged[5]. The *prog.c* file will be removed; it can be retrieved using *get*.

5.3. When to make deltas

It is probably unwise to make a delta before every recompilation or test; otherwise, you tend to get a lot of deltas with comments like "fixed compilation problem in previous delta" or "fixed botch in 1.3". However, it is very important to delta everything before installing a module for general use. A good technique is to edit the files you need, make all necessary changes and tests, compiling and editing as often as necessary without making deltas. When you are satisfied that you have a working version, delta everything being edited, re-get them, and recompile everything.

5.4. What's going on: the info command

To find out what files where being edited, you can use:

```
sccs info
```

to print out all the files being edited and other information such as the name of the user who did the edit. Also, the command:

```
sccs check
```

is nearly equivalent to the *info* command, except that it is silent if nothing is being edited, and returns non-zero exit status if anything is being edited; it can be used in an "install" entry in a makefile to abort the install if anything has not been properly deltaed.

If you know that everything being edited should be deltaed, you can use:

```
sccs delta `sccs tell`
```

The *tell* command is similar to *info* except that only the names of files being edited are output, one per line.

All of these commands take a **-b** flag to ignore "branches" (alternate versions, described later) and the **-u** flag to only give files being edited by you. The **-u** flag takes an optional *user* argument, giving only files being edited by that user. For example,

```
sccs info -ujohn
```

gives a listing of files being edited by john.

5.5. ID keywords

Id keywords can be inserted into your file that will be expanded automatically by *get*. For example, a line such as:

```
static char SccsId[] = "%W%\t%G%";
```

will be replaced with something like:

---

[4]Yes, this is a stupid default.

[5]Changes to a line are counted as a line deleted and a line inserted.

```
static char SccsId[] = "@(#)prog.c      1.2      08/29/80";
```

This tells you the name and version of the source file and the time the delta was created. The string "@(#)" is a special string which signals the beginning of an SCCS Id keyword.

5.5.1. The what command

To find out what version of a program is being run, use:

```
sccs what prog.c /usr/bin/prog
```

which will print all strings it finds that begin with "@(#)". This works on all types of files, including binaries and libraries. For example, the above command will output something like:

```
prog.c:
        prog.c  1.2     08/29/80
/usr/bin/prog:
        prog.c  1.1     02/05/79
```

From this I can see that the source that I have in prog.c will not compile into the same version as the binary in /usr/bin/prog.

5.5.2. Where to put id keywords

ID keywords can be inserted anywhere, including in comments, but Id Keywords that are compiled into the object module are especially useful, since it lets you find out what version of the object is being run, as well as the source. However, there is a cost: data space is used up to store the keywords, and on small address space machines this may be prohibitive.

When you put id keywords into header files, it is important that you assign them to different variables. For example, you might use:

```
static char AccessSid[] = "%W%      %G%";
```

in the file *access.h* and:

```
static char OpsysSid[] = "%W%      %G%";
```

in the file *opsys.h*. Otherwise, you will get compilation errors because "SccsId" is redefined. The problem with this is that if the header file is included by many modules that are loaded together, the version number of that header file is included in the object module many times; you may find it more to your taste to put id keywords in header files in comments.

5.6. Keeping SID's consistent across files

With some care, it is possible to keep the SID's consistent in multi-file systems. The trick here is to always *edit* all files at once. The changes can then be made to whatever files are necessary and then all files (even those not changed) are redeltaed. This can be done fairly easily by just specifying the name of the directory that the SCCS files are in:

```
sccs edit SCCS
```

which will *edit* all files in that directory. To make the delta, use:

```
sccs delta SCCS
```

You will be prompted for comments only once.

5.7. Creating new releases

When you want to create a new release of a program, you can specify the release number you want to create on the *edit* command. For example:

```
sccs edit -r2 prog.c
```

will cause the next delta to be in release two (that is, it will be numbered 2.1). Future deltas will automatically be in release two. To change the release number of an entire system, use:

```
sccs edit -r2 SCCS
```

6. Restoring Old Versions

6.1. Reverting to old versions

Suppose that after delta 1.2 was stable you made and released a delta 1.3. But this introduced a bug, so you made a delta 1.4 to correct it. But 1.4 was still buggy, and you decided you wanted to go back to the old version. You could revert to delta 1.2 by choosing the SID in a get:

```
sccs get -r1.2 prog.c
```

This will produce a version of *prog.c* that is delta 1.2 that can be reinstalled so that work can proceed.

In some cases you don't know what the SID of the delta you want is. However, you can revert to the version of the program that was running as of a certain date by using the **-c** (cutoff) flag. For example,

```
sccs get -c800722120000 prog.c
```

will retrieve whatever version was current as of July 22, 1980 at 12:00 noon. Trailing components can be stripped off (defaulting to their highest legal value), and punctuation can be inserted in the obvious places; for example, the above line could be equivalently stated:

```
sccs get -c"80/07/22 12:00:00" prog.c
```

6.2. Selectively deleting old deltas

Suppose that you later decided that you liked the changes in delta 1.4, but that delta 1.3 should be removed. You could do this by *excluding* delta 1.3:

```
sccs edit -x1.3 prog.c
```

When delta 1.5 is made, it will include the changes made in delta 1.4, but will exclude the changes made in delta 1.3. You can exclude a range of deltas using a dash. For example, if you want to get rid of 1.3 and 1.4 you can use:

```
sccs edit -x1.3-1.4 prog.c
```

which will exclude all deltas from 1.3 to 1.4. Alternatively,

```
sccs edit -x1.3-1 prog.c
```

will exclude a range of deltas from 1.3 to the current highest delta in release 1.

In certain cases when using **-x** (or **-i**; see below) there will be conflicts between versions; for example, it may be necessary to both include and delete a particular line. If this happens, SCCS always prints out a message telling the range of lines effected; these lines should then be examined very carefully to see if the version SCCS got is ok.

Since each delta (in the sense of "a set of changes") can be excluded at will, that this makes it most useful to put each semantically distinct change into its own delta.

7. Auditing Changes

7.1. The prt command

When you created a delta, you presumably gave a reason for the delta to the "comments?" prompt. To print out these comments later, use:

```
sccs prt prog.c
```

This will produce a report for each delta of the SID, time and date of creation, user who created the delta, number of lines inserted, deleted, and unchanged, and the comments associated with the delta. For example, the output of the above command might be:

```
D 1.2  80/08/29 12:35:31    bill    2       1       00005/00003/00084
removed "-q" option

D 1.1  79/02/05 00:19:31    eric    1       0       00087/00000/00000
date and time created 80/06/10 00:19:31 by eric
```

7.2. Finding why lines were inserted

To find out why you inserted lines, you can get a copy of the file with each line preceded by the SID that created it:

```
sccs get -m prog.c
```

You can then find out what this delta did by printing the comments using *prt*.

To find out what lines are associated with a particular delta (*e.g.*, 1.3), use:

```
sccs get -m -p prog.c | grep ^1.3'
```

The **-p** flag causes SCCS to output the generated source to the standard output rather than to a file.

7.3. Finding what changes you have made

When you are editing a file, you can find out what changes you have made using:

```
sccs diffs prog.c
```

Most of the "diff" flags can be used. To pass the **-c** flag, use **-C**.

To compare two versions that are in deltas, use:

```
sccs sccsdiff -r1.3 -r1.6 prog.c
```

to see the differences between delta 1.3 and delta 1.6.

8. Shorthand Notations

There are several sequences of commands that get executed frequently. *Sccs* tries to make it easy to do these.

8.1. Delget

A frequent requirement is to make a delta of some file and then get that file. This can be done by using:

```
sccs delget prog.c
```

which is entirely equivalent to using:

```
sccs delta prog.c
sccs get prog.c
```

The "deledit" command is equivalent to "delget" except that the "edit" command is used instead of the "get" command.

8.2. Fix

Frequently, there are small bugs in deltas, e.g., compilation errors, for which there is no reason to maintain an audit trail. To *replace* a delta, use:

```
sccs fix -r1.4 prog.c
```

This will get a copy of delta 1.4 of prog.c for you to edit and then delete delta 1.4 from the SCCS file. When you do a delta of prog.c, it will be delta 1.4 again. The -r flag must be specified, and the delta that is specified must be a leaf delta, i.e., no other deltas may have been made subsequent to the creation of that delta.

8.3. Unedit

If you found you edited a file that you did not want to edit, you can back out by using:

```
sccs unedit prog.c
```

8.4. The -d flag

If you are working on a project where the SCCS code is in a directory somewhere, you may be able to simplify things by using a shell alias. For example, the alias:

```
alias syssccs sccs -d/usr/src
```

will allow you to issue commands such as:

```
syssccs edit cmd/who.c
```

which will look for the file "/usr/src/cmd/SCCS/who.c". The file "who.c" will always be created in your current directory regardless of the value of the -d flag.

9. Using SCCS on a Project

Working on a project with several people has its own set of special problems. The main problem occurs when two people modify a file at the same time. SCCS prevents this by locking an s-file while it is being edited.

As a result, files should not be reserved for editing unless they are actually being edited at the time, since this will prevent other people on the project from making necessary changes. For example, a good scenario for working might be:

```
sccs edit a.c g.c t.c
vi a.c g.c t.c
# do testing of the (experimental) version
sccs delget a.c g.c t.c
sccs info
# should respond "Nothing being edited"
make install
```

As a general rule, all source files should be deltaed before installing the program for general use. This will insure that it is possible to restore any version in use at any time.

10. Saving Yourself

10.1. Recovering a munged edit file

Sometimes you may find that you have destroyed or trashed a file that you were trying

to edit[6]. Unfortunately, you can't just remove it and re-*edit* it; SCCS keeps track of the fact that someone is trying to edit it, so it won't let you do it again. Neither can you just get it using *get*, since that would expand the Id keywords. Instead, you can say:

```
sccs get -k prog.c
```

This will not expand the Id keywords, so it is safe to do a delta with it.

Alternately, you can *unedit* and *edit* the file.

10.2. Restoring the s-file

In particularly bad circumstances, the SCCS file itself may get munged. The most common way this happens is that it gets edited. Since SCCS keeps a checksum, you will get errors every time you read the file. To fix this checksum, use:

```
sccs admin -z prog.c
```

11. Using the Admin Command

There are a number of parameters that can be set using the *admin* command. The most interesting of these are flags. Flags can be added by using the **-f** flag. For example:

```
sccs admin -fdl prog.c
```

sets the "d" flag to the value "1". This flag can be deleted by using:

```
sccs admin -dd prog.c
```

The most useful flags are:

b Allow branches to be made using the -b flag to *edit*.

d*SID* Default SID to be used on a *get* or *edit*. If this is just a release number it constrains the version to a particular release only.

i Give a fatal error if there are no Id Keywords in a file. This is useful to guarantee that a version of the file does not get merged into the s-file that has the Id Keywords inserted as constants instead of internal forms.

y The "type" of the module. Actually, the value of this flag is unused by SCCS except that it replaces the **%Y%** keyword.

The **-t***file* flag can be used to store descriptive text from *file*. This descriptive text might be the documentation or a design and implementation document. Using the **-t** flag insures that if the SCCS file is sent, the documentation will be sent also. If *file* is omitted, the descriptive text is deleted. To see the descriptive text, use "prt -t".

The *admin* command can be used safely any number of times on files. A file need not be gotten for *admin* to work.

12. Maintaining Different Versions (Branches)

Sometimes it is convenient to maintain an experimental version of a program for an extended period while normal maintenance continues on the version in production. This can be done using a "branch." Normally deltas continue in a straight line, each depending on the delta before. Creating a branch "forks off" a version of the program.

The ability to create branches must be enabled in advance using:

```
sccs admin -fb prog.c
```

The **-fb** flag can be specified when the SCCS file is first created.

[6]Or given up and decided to start over.

12.1. Creating a branch

To create a branch, use:

```
sccs edit -b prog.c
```

This will create a branch with (for example) SID 1.5.1.1. The deltas for this version will be numbered 1.5.1.*n*.

12.2. Getting from a branch

Deltas in a branch are normally not included when you do a get. To get these versions, you will have to say:

```
sccs get -r1.5.1 prog.c
```

12.3. Merging a branch back into the main trunk

At some point you will have finished the experiment, and if it was successful you will want to incorporate it into the release version. But in the meantime someone may have created a delta 1.6 that you don't want to lose. The commands:

```
sccs edit -i1.5.1.1-1.5.1 prog.c
sccs delta prog.c
```

will merge all of your changes into the release system. If some of the changes conflict, get will print an error; the generated result should be carefully examined before the delta is made.

12.4. A more detailed example

The following technique might be used to maintain a different version of a program. First, create a directory to contain the new version:

```
mkdir ../newxyz
cd ../newxyz
```

Edit a copy of the program on a branch:

```
sccs -d../xyz edit prog.c
```

When using the old version, be sure to use the **-b** flag to info, check, tell, and clean to avoid confusion. For example, use:

```
sccs info -b
```

when in the directory "xyz".

If you want to save a copy of the program (still on the branch) back in the s-file, you can use:

```
sccs -d../xyz deledit prog.c
```

which will do a delta on the branch and reedit it for you.

When the experiment is complete, merge it back into the s-file using delta:

```
sccs -d../xyz delta prog.c
```

At this point you must decide whether this version should be merged back into the trunk (*i.e.* the default version), which may have undergone changes. If so, it can be merged using the **-i** flag to *edit* as described above.

12.5. A warning

Branches should be kept to a minimum. After the first branch from the trunk, SID's are assigned rather haphazardly, and the structure gets complex fast.

13. Using SCCS with Make

SCCS and make can be made to work together with a little care. A few sample makefiles for common applications are shown.

There are a few basic entries that every makefile ought to have. These are:

a.out	(or whatever the makefile generates.) This entry regenerates whatever this makefile is supposed to regenerate. If the makefile regenerates many things, this should be called "all" and should in turn have dependencies on everything the makefile can generate.
install	Moves the objects to the final resting place, doing any special *chmod*'s or *ranlib*'s as appropriate.
sources	Creates all the source files from SCCS files.
clean	Removes all files from the current directory that can be regenerated from SCCS files.
print	Prints the contents of the directory.

The examples shown below are only partial examples, and may omit some of these entries when they are deemed to be obvious.

The *clean* entry should not remove files that can be regenerated from the SCCS files. It is sufficiently important to have the source files around at all times that the only time they should be removed is when the directory is being mothballed. To do this, the command:

```
sccs clean
```

can be used. This will remove all files for which an s-file exists, but which is not being edited.

13.1. To maintain single programs

Frequently there are directories with several largely unrelated programs (such as simple commands). These can be put into a single makefile:

```
LDFLAGS= -i -s

prog: prog.o
	$(CC) $(LDFLAGS) -o prog prog.o
prog.o: prog.c prog.h

example: example.o
	$(CC) $(LDFLAGS) -o example example.o
example.o: example.c

.DEFAULT:
	sccs get $<
```

The trick here is that the .DEFAULT rule is called every time something is needed that does not exist, and no other rule exists to make it. The explicit dependency of the **.o** file on the **.c** file is important. Another way of doing the same thing is:

```
SRCS= prog.c prog.h example.c
LDFLAGS= -i -s
prog: prog.o
	$(CC) $(LDFLAGS) -o prog prog.o
prog.o: prog.h
example: example.o
	$(CC) $(LDFLAGS) -o example example.o
sources: $(SRCS)
$(SRCS):
	sccs get $@
```

There are a couple of advantages to this approach: (1) the explicit dependencies of the .o on the .c files are not needed, (2) there is an entry called "sources" so if you want to get all the sources you can just say "make sources", and (3) the makefile is less likely to do confusing things since it won't try to *get* things that do not exist.

13.2. To maintain a library

Libraries that are largely static are best updated using explicit commands, since *make* doesn't know about updating them properly. However, libraries that are in the process of being developed can be handled quite adequately. The problem is that the .o files have to be kept out of the library as well as in the library.

```
# configuration information
OBJS= a.o b.o c.o d.o
SRCS= a.c b.c c.c d.s x.h y.h z.h
TARG=	/usr/lib
# programs
GET= sccs get
REL=
AR=	-ar
RANLIB=	ranlib
lib.a: $(OBJS)
	$(AR) rvu lib.a $(OBJS)
	$(RANLIB) lib.a
install: lib.a
	sccs check
	cp lib.a $(TARG)/lib.a
	$(RANLIB) $(TARG)/lib.a
sources: $(SRCS)
$(SRCS):
	$(GET) $(REL) $@
print: sources
	pr *.h *.[cs]
clean:
	rm -f *.o
	rm -f core a.out $(LIB)
```

The "$(REL)" in the get can be used to get old versions easily; for example:

```
make b.o REL=-r1.3
```

The *install* entry includes the line "sccs check" before anything else. This guarantees that all the s-files are up to date (*i.e.*, nothing is being edited), and will abort the *make* if this condition is not met.

13.3. To maintain a large program

```
OBJS= a.o b.o c.o d.o
SRCS= a.c b.c c.y d.s x.h y.h z.h
GET= sccs get
REL=
a.out: $(OBJS)
        $(CC) $(LDFLAGS) $(OBJS) $(LIBS)
sources: $(SRCS)
$(SRCS):
        $(GET) $(REL) $@
```

(The *print* and *clean* entries are identical to the previous case.) This makefile requires copies of the source and object files to be kept during development. It is probably also wise to include lines of the form:

```
a.o: x.h y.h
b.o: z.h
c.o: x.h y.h z.h
z.h: x.h
```

so that modules will be recompiled if header files change.

Since *make* does not do transitive closure on dependencies, you may find in some makefiles lines like:

```
z.h: x.h
        touch z.h
```

This would be used in cases where file z.h has a line:

```
#include "x.h"
```

in order to bring the mod date of z.h in line with the mod date of x.h. When you have a makefile such as above, the *touch* command can be removed completely; the equivalent effect will be achieved by doing an automatic *get* on z.h.

14. Further Information

The *SCCS/PWB User's Manual* gives a deeper description of how to use SCCS. Of particular interest are the numbering of branches, the l-file, which gives a description of what deltas were used on a get, and certain other SCCS commands.

The SCCS manual pages are a good last resort. These should be read by software managers and by people who want to know everything about everything.

Both of these documents were written without the *sccs* front end in mind, so most of the examples are slightly different from those in this document.

Quick Reference

1. Commands

The following commands should all be preceded with "sccs". This list is not exhaustive; for more options see *Further Information.*

get — Gets files for compilation (not for editing). Id keywords are expanded.

- -r*SID* Version to get.
- -p Send to standard output rather than to the actual file.
- -k Don't expand id keywords.
- -i*list* List of deltas to include.
- -x*list* List of deltas to exclude.
- -m Precede each line with SID of creating delta.
- -c*date* Don't apply any deltas created after *date.*

edit — Gets files for editing. Id keywords are not expanded. Should be matched with a *delta* command.

- -r*SID* Same as *get.* If *SID* specifies a release that does not yet exist, the highest numbered delta is retrieved and the new delta is numbered with *SID.*
- -b Create a branch.
- -i*list* Same as *get.*
- -x*list* Same as *get.*

delta — Merge a file gotten using *edit* back into the s-file. Collect comments about why this delta was made.

unedit — Remove a file that has been edited previously without merging the changes into the s-file.

prt — Produce a report of changes.

- -t Print the descriptive text.
- -e Print (nearly) everything.

info — Give a list of all files being edited.

- -b Ignore branches.
- -u[*user*] Ignore files not being edited by *user.*

check — Same as *info*, except that nothing is printed if nothing is being edited and exit status is returned.

tell — Same as *info*, except that one line is produced per file being edited containing only the file name.

clean — Remove all files that can be regenerated from the s-file.

what — Find and print id keywords.

admin — Create or set parameters on s-files.

- -i*file* Create, using *file* as the initial contents.
- -z Rebuild the checksum in case the file has been trashed.

–f*flag* Turn on the *flag*.

–d*flag* Turn off (delete) the *flag*.

–t*file* Replace the descriptive text in the s-file with the contents of *file*. If *file* is omitted, the text is deleted. Useful for storing documentation or "design & implementation" documents to insure they get distributed with the s-file.

Useful flags are:

b Allow branches to be made using the –b flag to *edit*.

d*SID* Default SID to be used on a *get* or *edit*.

i Cause "No Id Keywords" error message to be a fatal error rather than a warning.

t The module "type"; the value of this flag replaces the **%Y%** keyword.

fix Remove a delta and reedit it.

delget Do a *delta* followed by a *get*.

deledit Do a *delta* followed by an *edit*.

2. Id Keywords

%Z% Expands to "@(#)" for the *what* command to find.

%M% The current module name, *e.g.*, "prog.c".

%I% The highest SID applied.

%W% A shorthand for "%Z%%M% <tab> %I%".

%G% The date of the delta corresponding to the "%I%" keyword.

%R% The current release number, *i.e.*, the first component of the "%I%" keyword.

%Y% Replaced by the value of the t flag (set by *admin*).

Yacc: Yet Another Compiler-Compiler

Stephen C. Johnson

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

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

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

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

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

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

0: Introduction

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a *parser*, calls the user-supplied low-level input routine (the *lexical analyzer*) to pick up the basic items (called *tokens*) from the input stream. These tokens are organized according to the input structure rules, called *grammar rules*; when one of these rules has been recognized, then user code supplied for this rule, an *action*, is invoked; actions have the ability to return values and make use of the values of other actions.

Yacc is written in a portable dialect of C[1] and the actions, and output subroutine, are in C as well. Moreover, many of the syntactic conventions of Yacc follow C.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

```
date : month_name day ',' year ;
```

Here, *date*, *month_name*, *day*, and *year* represent structures of interest in the input process; presumably, *month_name*, *day*, and *year* are defined elsewhere. The comma "," is enclosed in single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

```
July 4, 1776
```

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a *terminal symbol*, while the structure recognized by the parser is called a *nonterminal symbol*. To avoid confusion, terminal symbols will usually be referred to as *tokens*.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

```
month_name : 'J' 'a' 'n' ;
month_name : 'F' 'e' 'b' ;

    . . .

month_name : 'D' 'e' 'c' ;
```

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and *month_name* would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond Yacc's ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a *month_name* was seen; in this case, *month_name* would be a token.

Literal characters such as "," must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is realively easy to add to the above example the rule

```
date : month '/' day '/' year ;
```

allowing

```
7 / 4 / 1776
```

as a synonym for

```
July 4, 1776
```

In most cases, this new rule could be "slipped in" to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, Yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self contradictory, or they may require a more powerful recognition mechanism than that available to Yacc. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While Yacc cannot handle all possible specifications, its power compares favorably with similar systems; moreover, the constructions which are difficult for Yacc to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid Yacc specifications for their input revealed errors of conception or design early in the program

development.

The theory underlying Yacc has been described elsewhere.[2,3,4] Yacc has been extensively used in numerous practical applications, including *lint*,[5] the Portable C Compiler,[6] and a system for typesetting mathematics.[7]

The next several sections describe the basic process of preparing a Yacc specification; Section 1 describes the preparation of grammar rules, Section 2 the preparation of the user supplied actions associated with these rules, and Section 3 the preparation of lexical analyzers. Section 4 describes the operation of the parser. Section 5 discusses various reasons why Yacc may be unable to produce a parser from a specification, and what to do about it. Section 6 describes a simple mechanism for handling operator precedences in arithmetic expressions. Section 7 discusses error detection and recovery. Section 8 discusses the operating environment and special features of the parsers Yacc produces. Section 9 gives some suggestions which should improve the style and efficiency of the specifications. Section 10 discusses some advanced topics, and Section 11 gives acknowledgements. Appendix A has a brief example, and Appendix B gives a summary of the Yacc input syntax. Appendix C gives an example using some of the more advanced features of Yacc, and, finally, Appendix D describes mechanisms and syntax no longer actively supported, but provided for historical continuity with older versions of Yacc.

1: Basic Specifications

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. In addition, for reasons discussed in Section 3, it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections: the *declarations*, *(grammar) rules*, and *programs*. The sections are separated by double percent "%%" marks. (The percent "%" is generally used in Yacc specifications as an escape character.)

In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal Yacc specification is

```
%%
rules
```

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* . . . */, as in C and PL/I.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A : BODY ;
```

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are Yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot ".", underscore "_", and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes "'". As in C, the backslash "\" is an escape character within literals, and all the C escapes are recognized. Thus

'\n' newline
'\r' return
'\'' single quote "'"
'\\' backslash "\"
'\t' tab
'\b' backspace
'\f' form feed
'\xxx' "xxx" in octal

For a number of technical reasons, the NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, the vertical bar "|" can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

```
A : B C D ;
A : E F ;
A : G ;
```

can be given to Yacc as

```
A : B C D
  | E F
  | G
  ;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

```
empty : ;
```

Names representing tokens must be declared; this is most simply done by writing

```
%token name1 name2 . . .
```

in the declarations section. (See Sections 3 , 5, and 6 for much more discussion). Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the *start symbol*, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the %start keyword:

```
%start symbol
```

The end of the input to the parser is signaled by a special token, called the *endmarker*. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the endmarker is seen; it *accepts* the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate; see section 3, below. Usually the endmarker represents some reasonably obvious I/O status, such as "end-of-file" or "end-of-record".

2: Actions

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces "{" and "}". For example,

```
A   :   '(' B ')'
            {   hello( 1, "abc" ); }
```

and

```
XXX:    YYY ZZZ
            {   printf("a message\n");
                flag = 25; }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol "dollar sign" "$" is used as a signal to Yacc in this context.

To return a value, the action normally sets the pseudo-variable "$$" to some value. For example, an action that does nothing but return the value 1 is

```
{ $$ = 1; }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, . . ., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

```
A   :   B C D ;
```

for example, then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, consider the rule

```
expr :   '(' expr ')' ;
```

The value returned by this rule is usually the value of the *expr* in parentheses. This can be indicated by

```
expr :   '(' expr ')'   { $$ = $2 ; }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form

```
A   :   B ;
```

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. Yacc permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

```
A   :   B
            { $$ = 1; }
        C
            {  x = $2;  y = $3; }
    ;
```

the effect is to set x to 1, and y to the value returned by C.

Actions that do not terminate a rule are actually handled by Yacc by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. Yacc actually treats the above example as if it had been written:

```
$ACT    :    /* empty */
             { $$ = 1; }
    ;

A   :    B $ACT C
             { x = $2;  y = $3; }
    ;
```

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function *node*, written so that the call

```
node( L, n1, n2 )
```

creates a node with label L, and descendants n1 and n2, and returns the index of the newly created node. Then parse tree can be built by supplying actions such as:

```
expr :    expr '+' expr
             { $$ = node( '+', $1, $3 ); }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks "%{" and "%}". These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{  int variable = 0;  %}
```

could be placed in the declarations section, making *variable* accessible to all of the actions. The Yacc parser uses only names beginning in "yy"; the user should avoid such names.

In these examples, all the values are integers: a discussion of values of other types will be found in Section 10.

3: Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called *yylex*. The function returns an integer, the *token number*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable *yylval*.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by Yacc, or chosen by the user. In either case, the "# define" mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the Yacc specification file. The relevant portion of the lexical analyzer might look like:

```
yylex(){
        extern int yylval;
        int c;
        . . .
        c = getchar();
        . . .
        switch( c ) {
                . . .
        case '0':
        case '1':
          . . .
        case '9':
                yylval = c-'0';
                return( DIGIT );
                . . .
                }
        . . .
```

The intent is to return a token number of DIGIT, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT will be defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of token names *if* or *while* will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name *error* is reserved for error handling, and should not be used naively (see Section 7).

As mentioned above, the token numbers may be chosen by Yacc or by the user. In the default situation, the numbers are chosen by Yacc. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

To assign a token number to a token (including literals), the first appearance of the token name or literal *in the declarations section* can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user; thus, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the *Lex* program developed by Mike Lesk.[8] These lexical analyzers are designed to work in close harmony with Yacc parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. Lex can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) which do not fit any theoretical framework, and whose lexical analyzers must be crafted by hand.

4: How the Parser Works

Yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the *lookahead* token). The *current state* is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called *shift*, *reduce*, *accept*, and *error*. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls *yylex* to obtain the next token.
2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

The *shift* action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

```
IF    shift 34
```

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

The *reduce* action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a ".") is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

```
.     reduce 18
```

refers to *grammar rule* 18, while the action

```
IF    shift 34
```

refers to *state* 34.

Suppose the rule being reduced is

```
A  :  x y z  ;
```

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped equals the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack while recognizing x, y, and z, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left hand symbol and an ordinary shift of a token, however, so this action is called a *goto* action. In particular, the lookahead token is cleared by a shift, and is not affected by a goto. In any case, the uncovered state contains an entry such as:

```
A     goto 20
```

causing state 20 to be pushed onto the stack, and become the current state.

In effect, the reduce action "turns back the clock" in the parse, popping the states off the stack to go back to the state where the right hand side of the rule was first seen. The parser then behaves

as if it had seen the left side at that time. If the right hand side of the rule is empty, no states are popped off of the stack: the uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable *yylval* is copied onto the value stack. After the return from the user code, the reduction is carried out. When the *goto* action is done, the external variable *yyval* is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The *accept* action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The *error* action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The parser reports an error, and attempts to recover the situation and resume parsing: the error recovery (as opposed to the detection of error) will be covered in Section 7.

It is time for an example! Consider the specification

```
%token DING DONG DELL
%%
rhyme   :   sound place
        ;
sound   :   DING DONG
        ;
place   :   DELL
        ;
```

When Yacc is invoked with the -v option, a file called *y.output* is produced, with a human-readable description of the parser. The *y.output* file corresponding to the above grammar (with some statistics stripped off the end) is:

```
state 0
        $accept : _rhyme $end

        DING shift 3
        . error

        rhyme goto 1
        sound goto 2

state 1
        $accept : rhyme_$end

        $end accept
        . error

state 2
        rhyme : sound_place

        DELL shift 5
        . error

        place goto 4

state 3
        sound : DING_DONG

        DONG shift 6
        . error

state 4
        rhyme : sound place_  (1)

        . reduce 1

state 5
        place : DELL_  (3)

        . reduce 3

state 6
        sound : DING DONG_  (2)

        . reduce 2
```

Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The _ character is used to indicate what has been seen, and what is yet to come, in each rule. Suppose the input is

```
DING DONG DELL
```

It is instructive to follow the steps of the parser while processing this input.

Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, *DING*, is read, becoming the lookahead token. The action in state 0 on *DING* is is "shift 3", so state 3 is pushed onto the stack, and the lookahead token is cleared. State 3 becomes the current state. The next token, *DONG*, is read,

becoming the lookahead token. The action in state 3 on the token *DONG* is "shift 6", so state 6 is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the lookahead, the parser reduces by rule 2.

```
sound : DING DONG
```

This rule has two symbols on the right hand side, so two states, 6 and 3, are popped off of the stack, uncovering state 0. Consulting the description of state 0, looking for a goto on *sound*,

```
sound    goto 2
```

is obtained; thus state 2 is pushed onto the stack, becoming the current state.

In state 2, the next token, *DELL*, must be read. The action is "shift 5", so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the lookahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on *place*, the left side of rule 3, is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on *rhyme* causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained, indicated by "$end" in the *y.output* file. The action in state 1 when the endmarker is seen is to accept, successfully ending the parse.

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

5: Ambiguity and Conflicts

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

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

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

```
expr - expr - expr
```

the rule allows this input to be structured as either

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

or as

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

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

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

```
expr - expr - expr
```

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

```
expr - expr
```

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

```
        - expr
```

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

Alternatively, when the parser has seen

```
        expr - expr
```

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

```
        expr - expr - expr
```

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

```
        expr - expr
```

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

```
        expr - expr
```

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

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

Yacc invokes two disambiguating rules by default:

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

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

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

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

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

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

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

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

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

can be structured according to these rules in two ways:

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

or

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

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

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

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

```
IF ( C1 ) stat
```

and then read the remaining input,

```
ELSE S2
```

and reduce

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

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

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

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

can be reduced by the if-else rule to get

```
IF ( C1 ) stat
```

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

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

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

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

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

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

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

state 23

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

      ELSE    shift 45
              reduce 18
```

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

```
IF ( cond ) stat
```

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

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

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

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

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

6: Precedence

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

```
expr : expr OP expr
```

and

```
expr : UNARY expr
```

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

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

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

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

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

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

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

%%

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

might be used to structure the input

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

as follows:

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

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

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

%%

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

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

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

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

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

7: Error Handling

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

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

To allow the user some control over this process, Yacc provides a simple, but reasonably general, feature. The token name "error" is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. The parser pops its stack until it enters a state where the token "error" is legal. It then behaves as if the token "error" were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form

```
stat  :     error
```

would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement, and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general, but difficult to control. Somewhat easier are rules such as

```
stat  :     error  ';'
```

Here, when there is an error, the parser attempts to skip over the statement, but will do so by skipping to the next ';'. All tokens after the error and before the next ';' cannot be shifted, and are discarded. When the ';' is seen, this rule will be reduced, and any "cleanup" action associated with it performed.

Another form of error rule arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error rule might be

```
input:      error  '\n'  {  printf( "Reenter last line: " );  }  input
                    {     $$  =  $4;  }
```

There is one potential difficulty with this approach; the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example is better written

```
input:      error  '\n'
                    {     yyerrok;
                          printf( "Reenter last line: " );   }
            input
                    {     $$  =  $4;  }
        ;
```

As mentioned above, the token seen immediately after the "error" symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after error were to call some sophisticated resynchronization routine, supplied by the user, that attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by yylex would presumably be the first token in a legal statement; the old, illegal token must be discarded, and the error state reset. This could be done by a rule like

```
stat :  error
            {   resynch();
                yyerrok ;
                yyclearin ;  }
    ;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors; moreover, the user can get control to deal with the error actions required by other portions of the program.

8: The Yacc Environment

When the user inputs a specification to Yacc, the output is a file of C programs, called *y.tab.c* on most systems (due to local file system conventions, the names may differ from installation to installation). The function produced by Yacc is called *yyparse*; it is an integer valued function. When it is called, it in turn repeatedly calls *yylex*, the lexical analyzer supplied by the user (see Section 3) to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) *yyparse* returns the value 1, or the lexical analyzer returns the endmarker token and the parser accepts. In this case, *yyparse* returns the value 0.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C program, a program called *main* must be defined, that eventually calls *yyparse*. In addition, a routine called *yyerror* prints a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using Yacc, a library has been provided with default versions of *main* and *yyerror*. The name of this library is system dependent; on many systems the library is accessed by a **–ly** argument to the loader. To show the triviality of these default programs, the source is given below:

```
main(){
    return( yyparse() );
    }
```

and

```
# include <stdio.h>

yyerror(s) char *s; {
    fprintf( stderr, "%s\n", s );
    }
```

The argument to *yyerror* is a string containing an error message, usually the string "syntax error". The average application will want to do better than this. Ordinarily, the program should keep track of the input line number, and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the lookahead token number at the time the error was detected; this may be of some interest in giving better diagnostics. Since the *main* program is probably supplied by the user (to read arguments, etc.) the Yacc library is useful only in small projects, or in the earliest stages of larger ones.

The external integer variable *yydebug* is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

9: Hints for Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following style hints owe much to Brian Kernighan.

a. Use all capital letters for token names, all lower case letters for nonterminal names. This rule comes under the heading of "knowing who to blame when things go wrong."

b. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.

c. Put all rules with the same left hand side together. Put the left hand side in only once, and let all following rules begin with a vertical bar.

d. Put a semicolon only after the last rule with a given left hand side, and put the semicolon on a separate line. This allows new rules to be easily added.

e. Indent rule bodies by two tab stops, and action bodies by three tab stops.

The example in Appendix A is written following this style, as are the examples in the text of this paper (where space permits). The user must make up his own mind about these stylistic questions; the central problem, however, is to make the rules visible through the morass of action code.

Left Recursion

The algorithm used by the Yacc parser encourages so called "left recursive" grammar rules: rules of the form

```
name:     name rest_of_rule ;
```

These rules frequently arise when writing specifications of sequences and lists:

```
list   :     item
       |     list ',' item
       ;
```

and

```
seq    :     item
       |     seq item
       ;
```

In each of these cases, the first rule will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as

```
seq    :     item
       |     item seq
       ;
```

the parser would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion wherever reasonable.

It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
seq    :     /* empty */
       |     seq item
       ;
```

Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if Yacc is asked to decide which empty sequence it has seen, when it hasn't seen enough to know!

Lexical Tie-ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names might be entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of 0 or more declarations, followed by 0 or more statements. Consider:

```
%{
        int dflag;
%}
 ... other declarations ...

%%

prog :      decls  stats
        ;

decls :     /* empty */
                    {     dflag = 1; }
        |   decls  declaration
        ;

stats :     /* empty */
                    {     dflag = 0; }
        |   stats  statement
        ;

   ... other rules ...
```

The flag *dflag* is now 0 when reading statements, and 1 when reading declarations, *except for the first token in the first statement.* This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of "backdoor" approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Reserved Words

Some programming languages permit the user to use words like "if", which are normally reserved, as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of Yacc; it is difficult to pass information to the lexical analyzer telling it "this instance of 'if' is a keyword, and that instance is a variable". The user can make a stab at it, using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be *reserved*; that is, be forbidden for use as variable names. There are powerful stylistic reasons for preferring this, anyway.

10: Advanced Topics

This section discusses a number of advanced features of Yacc.

Simulating Error and Accept in Actions

The parsing actions of error and accept can be simulated in an action by use of macros YYACCEPT and YYERROR. YYACCEPT causes *yyparse* to return the value 0; YYERROR causes the parser to behave as if the current input symbol had been a syntax error; *yyerror* is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple endmarkers or context-sensitive syntax checking.

Accessing Values in Enclosing Rules.

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit, but in this case the digit may be 0 or negative. Consider

```
sent :    adj noun verb adj noun
              { look at the sentence . . . }
     ;

adj  :    THE      {    $$ = THE; }
     |    YOUNG    {    $$ = YOUNG; }
     . . .
     ;

noun :    DOG
              {    $$ = DOG; }
     |    CRONE
              {    if( $0 == YOUNG ){
                       printf( "what?\n" );
                       }
                   $$ = CRONE;
                   }
     ;
     . . .
```

In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG. Obviously, this is only possible when a great deal is known about what might precede the symbol *noun* in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism will save a great deal of trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. Yacc can also support values of other types, including structures. In addition, Yacc keeps track of the types, and inserts appropriate union member names so that the resulting parser will be strictly type checked. The Yacc value stack (see Section 4) is declared to be a *union* of the various types of values desired. The user declares the union, and associates union member names to each token and nonterminal symbol having a value. When the value is referenced through a $$ or $n construction, Yacc will automatically insert the appropriate union name, so that no unwanted conversions will take place. In addition, type checking commands such as *Lint*[5] will be far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union; this must be done by the user since other programs, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where Yacc can not easily determine the type.

To declare the union, the user includes in the declaration section:

```
%union {
    body of union ...
    }
```

This declares the Yacc value stack, and the external variables *yylval* and *yyval*, to have type equal to this union. If Yacc was invoked with the **-d** option, the union declaration is copied onto the *y.tab.h* file. Alternatively, the union may be declared in a header file, and a typedef used to define the variable YYSTYPE to represent this union. Thus, the header file might also have said:

```
typedef union {
    body of union ...
    } YYSTYPE;
```

The header file must be included in the declarations section, by use of %{ and %}.

Once YYSTYPE is defined, the union member names must be associated with the various terminal and nonterminal names. The construction

```
< name >
```

is used to indicate a union member name. If this follows one of the keywords %token, %left, %right, and %nonassoc, the union member name is associated with the tokens listed. Thus, saying

```
%left <optype> '+' '-'
```

will cause any reference to values returned by these two tokens to be tagged with the union member name *optype*. Another keyword, %type, is used similarly to associate union member names with nonterminals. Thus, one might say

```
%type <nodetype> expr stat
```

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no *a priori* type. Similarly, reference to left context values (such as $0 – see the previous subsection) leaves Yacc with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name, between < and >, immediately after the first $. An example of this usage is

```
rule :    aaa { $<intval>$ = 3; } bbb
              {    fun( $<intval>2, $<other>0 ); }
    ;
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in Appendix C. The facilities in this subsection are not triggered until they are used: in particular, the use of %type will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of $n or $$ to refer to something with no defined type is diagnosed. If these facilities are not triggered, the Yacc value stack is used to hold *int*'s, as was true historically.

11: Acknowledgements

Yacc owes much to a most stimulating collection of users, who have goaded me beyond my inclination, and frequently beyond my ability, in their endless search for "one more feature". Their irritating unwillingness to learn how to do things my way has usually led to my doing things their way; most of the time, they have been right. B. W. Kernighan, P. J. Plauger, S. I. Feldman, C. Imagna, M. E. Lesk, and A. Snyder will recognize some of their ideas in the current version of Yacc. C. B. Haley contributed to the error recovery algorithm. D. M. Ritchie, B. W. Kernighan, and M. O. Harris helped translate this document into English. Al Aho also deserves special credit for bringing the mountain to Mohammed, and other favors.

References

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
2. A. V. Aho and S. C. Johnson, "LR Parsing," *Comp. Surveys*, vol. 6, no. 2, pp. 99-124, June 1974.
3. A. V. Aho, S. C. Johnson, and J. D. Ullman, "Deterministic Parsing of Ambiguous Grammars," *Comm. Assoc. Comp. Mach.*, vol. 18, no. 8, pp. 441-452, August 1975.
4. A. V. Aho and J. D. Ullman, *Principles of Compiler Design,* Addison-Wesley, Reading, Mass., 1977.
5. S. C. Johnson, "Lint, a C Program Checker," Comp. Sci. Tech. Rep. No. 65, 1978. Reprinted as PS1:9 in *UNIX Programmer's Manual*, Usenix Association, (1986).
6. S. C. Johnson, "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages*, pp. 97-104, January 1978.
7. B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," *Comm. Assoc. Comp. Mach.*, vol. 18, pp. 151-157, Bell Laboratories, Murray Hill, New Jersey, March 1975. Reprinted as USD:26 in *UNIX User's Manual*, Usenix Association, (1986).
8. M. E. Lesk, "Lex — A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey, October 1975. Reprinted as PS1:16 in *UNIX Programmer's Manual*, Usenix Association, (1986).

Appendix A: A Simple Example

This example gives the complete Yacc specification for a small desk calculator; the desk calculator has 26 registers, labeled "a" through "z", and accepts arithmetic expressions made up of the operators +, –, *, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression at the top level is an assignment, the value is not printed; otherwise it is. As in C, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a Yacc specification, the desk calculator does a reasonable job of showing how precedences and ambiguities are used, and demonstrating simple error recovery. The major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately, line by line. Note the way that decimal and octal integers are read in by the grammar rules; This job is probably better done by the lexical analyzer.

```
%{
# include <stdio.h>
# include <ctype.h>

int regs[26];
int base;

%}

%start list

%token DIGIT LETTER

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS    /* supplies precedence for unary minus */

%%    /* beginning of rules section */

list    :       /* empty */
        |       list stat '\n'
        |       list error '\n'
                        {       yyerrok; }
        ;

stat    :       expr
                        {       printf( "%d\n", $1 ); }
        |       LETTER '=' expr
                        {       regs[$1] = $3; }
        ;

expr    :       '(' expr ')'
                        {       $$ = $2; }
        |       expr '+' expr
                        {       $$ = $1 + $3; }
        |       expr '-' expr
                        {       $$ = $1 - $3; }
        |       expr '*' expr
                        {       $$ = $1 * $3; }
```

```
        |       expr '/' expr
                        {       $$ = $1 / $3; }
        |       expr '%' expr
                        {       $$ = $1 % $3; }
        |       expr '&' expr
                        {       $$ = $1 & $3; }
        |       expr '|' expr
                        {       $$ = $1 | $3; }
        |       '-' expr        %prec UMINUS
                        {       $$ = - $2; }
        |       LETTER
                        {       $$ = regs[$1]; }
        |       number
        ;

number:         DIGIT
                        {       $$ = $1;   base = ($1==0) ? 8 : 10; }
        |       number DIGIT
                        {       $$ = base * $1 + $2; }
        ;

%%      /* start of programs */

yylex() {               /* lexical analysis routine */
          /* returns LETTER for a lower case letter, yylval = 0 through 25 */
          /* return DIGIT for a digit, yylval = 0 through 9 */
          /* all other characters are returned immediately */

        int c;

        while( (c=getchar()) == ' ' ) {/* skip blanks */ }

        /* c is now nonblank */

        if( islower( c ) ) {
                yylval = c - 'a';
                return ( LETTER );
                }
        if( isdigit( c ) ) {
                yylval = c - '0';
                return( DIGIT );
                }
        return( c );
        }
```

Appendix B: Yacc Input Syntax

This Appendix has a description of the Yacc input syntax, as a Yacc specification. Context dependencies, etc., are not considered. Ironically, the Yacc input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule, immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier, and decide whether the next token (skipping blanks, newlines, comments, etc.) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERS, but never as part of C_IDENTIFIERs.

```
        /* grammar for the input to Yacc */

        /* basic entities */
%token  IDENTIFIER      /*  includes identifiers and literals */
%token  C_IDENTIFIER    /*  identifier (but not literal) followed by colon    */
%token  NUMBER                  /*  [0-9]+    */

        /* reserved words:  %type => TYPE, %left => LEFT, etc. */

%token  LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token  MARK  /* the %% mark */
%token  LCURL /* the %{ mark */
%token  RCURL /* the %} mark */

        /* ascii character literals stand for themselves */

%start  spec

%%

spec    :       defs MARK rules tail
        ;

tail    :       MARK { In this action, eat up the rest of the file }
        |       /* empty: the second MARK is optional */
        ;

defs    :       /* empty */
        |       defs def
        ;

def     :       START IDENTIFIER
        |       UNION { Copy union definition to output }
        |       LCURL { Copy C code to output file } RCURL
        |       ndefs rword tag nlist
        ;

rword   :       TOKEN
        |       LEFT
        |       RIGHT
        |       NONASSOC
```

```
        |       TYPE
        ;

tag     :       /* empty: union tag is optional */
        |       '<' IDENTIFIER '>'
        ;

nlist   :       nmno
        |       nlist nmno
        |       nlist ',' nmno
        ;

nmno    :       IDENTIFIER              /* NOTE: literal illegal with %type */
        |       IDENTIFIER NUMBER       /* NOTE: illegal with %type */
        ;

        /* rules section */

rules   :       C_IDENTIFIER rbody prec
        |       rules rule
        ;

rule    :       C_IDENTIFIER rbody prec
        |       '|' rbody prec
        ;

rbody   :       /* empty */
        |       rbody IDENTIFIER
        |       rbody act
        ;

act     :       '{' { Copy action, translate $$, etc. } '}'
        ;

prec    :       /* empty */
        |       PREC IDENTIFIER
        |       PREC IDENTIFIER act
        |       prec ';'
        ;
```

Appendix C: An Advanced Example

This Appendix gives an example of a grammar using some of the advanced features discussed in Section 10. The desk calculator example in Appendix A is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants, the arithmetic operations +, −, *, /, unary −, and = (assignment), and has 26 floating point variables, "a" through "z". Moreover, it also understands *intervals*, written

(x , y)

where x is less than or equal to y. There are 26 interval valued variables "A" through "Z" that may also be used. The usage is similar to that in Appendix A; assignments return no value, and print nothing, while expressions print the (floating or interval) value.

This example explores a number of interesting features of Yacc and C. Intervals are represented by a structure, consisting of the left and right endpoint values, stored as *double*'s. This structure is given a type name, INTERVAL, by using *typedef*. The Yacc value stack can also contain floating point scalars, and integers (used to index into the arrays holding the variable values). Notice that this entire strategy depends strongly on being able to assign structures and unions in C. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of YYERROR to handle error conditions: division by an interval containing 0, and an interval presented in the wrong order. In effect, the error recovery mechanism of Yacc is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (e.g. scalar or interval) of intermediate expressions. Note that a scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through Yacc: 18 Shift/Reduce and 26 Reduce/Reduce. The problem can be seen by looking at the two input lines:

2.5 + (3.5 − 4.)

and

2.5 + (3.5 , 4.)

Notice that the 2.5 is to be used in an interval valued expression in the second example, but this fact is not known until the "," is read; by this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator: one when the left operand is a scalar, and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflicts will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive, but not very general. If there were many kinds of expression types, instead of just two, the number of rules needed would increase dramatically, and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value, and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C library routine *atof* is used to do the actual conversion from a character string to a double precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar, provoking a syntax error in the parser, and thence error recovery.

```
%{

# include <stdio.h>
# include <ctype.h>

typedef struct interval {
     double lo, hi;
     } INTERVAL;

INTERVAL vmul(), vdiv();

double atof();

double dreg[ 26 ];
INTERVAL vreg[ 26 ];

%}

%start  lines

%union  {
     int ival;
     double dval;
     INTERVAL vval;
     }

%token <ival> DREG VREG /* indices into dreg, vreg arrays */

%token <dval> CONST        /* floating point constant */

%type <dval> dexp       /* expression */

%type <vval> vexp       /* interval expression */

     /* precedence information about the operators */

%left '+' '-'
%left '*' '/'
%left UMINUS      /* precedence for unary minus */

%%

lines :     /* empty */
     |      lines line
     ;

line :      dexp '\n'
                {     printf( "%15.8f\n", $1 ); }
     |      vexp '\n'
                {     printf( "(%15.8f , %15.8f )\n", $1.lo, $1.hi ); }
     |      DREG '=' dexp '\n'
                {     dreg[$1] = $3; }
     |      VREG '=' vexp '\n'
```

```
            {   vreg[$1] = $3; }
    |   error '\n'
            {   yyerrok; }
    ;

dexp :  CONST
    |   DREG
            {   $$ = dreg[$1]; }
    |   dexp '+' dexp
            {   $$ = $1 + $3; }
    |   dexp '-' dexp
            {   $$ = $1 - $3; }
    |   dexp '*' dexp
            {   $$ = $1 * $3; }
    |   dexp '/' dexp
            {   $$ = $1 / $3; }
    |   '-' dexp   %prec UMINUS
            {   $$ = -$2; }
    |   '(' dexp ')'
            {   $$ = $2; }
    ;

vexp :  dexp
            {   $$.hi = $$.lo = $1; }
    |   '(' dexp ',' dexp ')'
            {
            $$.lo = $2;
            $$.hi = $4;
            if( $$.lo > $$.hi ){
                printf( "interval out of order\n" );
                YYERROR;
                }
            }
    |   VREG
            {   $$ = vreg[$1];  }
    |   vexp '+' vexp
            {   $$.hi = $1.hi + $3.hi;
                $$.lo = $1.lo + $3.lo;  }
    |   dexp '+' vexp
            {   $$.hi = $1 + $3.hi;
                $$.lo = $1 + $3.lo;  }
    |   vexp '-' vexp
            {   $$.hi = $1.hi - $3.lo;
                $$.lo = $1.lo - $3.hi;  }
    |   dexp '-' vexp
            {   $$.hi = $1 - $3.lo;
                $$.lo = $1 - $3.hi;  }
    |   vexp '*' vexp
            {   $$ = vmul( $1.lo, $1.hi, $3 ); }
    |   dexp '*' vexp
            {   $$ = vmul( $1, $1, $3 ); }
    |   vexp '/' vexp
            {   if( dcheck( $3 ) ) YYERROR;
                $$ = vdiv( $1.lo, $1.hi, $3 ); }
```

```
    |   dexp '/' vexp
            {   if( dcheck( $3 ) ) YYERROR;
                $$ = vdiv( $1, $1, $3 ); }
    |   '-' vexp   %prec UMINUS
            {   $$.hi = -$2.lo;   $$.lo = -$2.hi;   }
    |   '(' vexp ')'
            {   $$ = $2; }
    ;

%%

# define BSZ 50      /* buffer size for floating point numbers */

    /* lexical analysis */

yylex(){
    register c;

    while( (c=getchar()) == ' ' ){ /* skip over blanks */ }

    if( isupper( c ) ){
        yylval.ival = c - 'A';
        return( VREG );
        }
    if( islower( c ) ){
        yylval.ival = c - 'a';
        return( DREG );
        }

    if( isdigit( c ) || c=='.' ){
        /* gobble up digits, points, exponents */

        char buf[BSZ+1], *cp = buf;
        int dot = 0, exp = 0;

        for( ; (cp-buf)<BSZ ; ++cp,c=getchar() ){

            *cp = c;
            if( isdigit( c ) ) continue;
            if( c == '.' ){
                if( dot++ || exp ) return( '.' );   /* will cause syntax error */
                continue;
                }

            if( c == 'e' ){
                if( exp++ ) return( 'e' );   /* will cause syntax error */
                continue;
                }

            /* end of number */
            break;
            }
        *cp = '\0';
        if( (cp-buf) >= BSZ ) printf( "constant too long: truncated\n" );
```

```
        else ungetc( c, stdin );  /* push back last char read */
        yylval.dval = atof( buf );
        return( CONST );
        }
    return( c );
    }

INTERVAL hilo( a, b, c, d ) double a, b, c, d; {
    /* returns the smallest interval containing a, b, c, and d */
    /* used by *, / routines */
    INTERVAL v;

    if( a>b ) { v.hi = a;  v.lo = b; }
    else { v.hi = b;  v.lo = a; }

    if( c>d ) {
        if( c>v.hi ) v.hi = c;
        if( d<v.lo ) v.lo = d;
        }
    else {
        if( d>v.hi ) v.hi = d;
        if( c<v.lo ) v.lo = c;
        }
    return( v );
    }

INTERVAL vmul( a, b, v ) double a, b;  INTERVAL v; {
    return( hilo( a*v.hi, a*v.lo, b*v.hi, b*v.lo ) );
    }

dcheck( v ) INTERVAL v; {
    if( v.hi >= 0. && v.lo <= 0. ){
        printf( "divisor interval contains 0.\n" );
        return( 1 );
        }
    return( 0 );
    }

INTERVAL vdiv( a, b, v ) double a, b;  INTERVAL v; {
    return( hilo( a/v.hi, a/v.lo, b/v.hi, b/v.lo ) );
    }
```

Appendix D: Old Features Supported but not Encouraged

This Appendix mentions synonyms and features which are supported for historical continuity, but, for various reasons, are not encouraged.

1. Literals may also be delimited by double quotes "".
2. Literals may be more than one character long. If all the characters are alphabetic, numeric, or _, the type number of the literal is defined, just as if the literal did not have the quotes around it. Otherwise, it is difficult to find the value for such literals.

   The use of multi-character literals is likely to mislead those unfamiliar with Yacc, since it suggests that Yacc is doing a job which must be actually done by the lexical analyzer.
3. Most places where % is legal, backslash "\" may be used. In particular, \\ is the same as %%, \left the same as %left, etc.
4. There are a number of other synonyms:

   %< is the same as %left
   %> is the same as %right
   %binary and %2 are the same as %nonassoc
   %0 and %term are the same as %token
   %= is the same as %prec

5. Actions may also have the form

   ={ . . . }

   and the curly braces can be dropped if the action is a single C statement.
6. C code between %{ and %} used to be permitted at the head of the rules section, as well as in the declaration section.

Lex – A Lexical Analyzer Generator

M. E. Lesk and E. Schmidt

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

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

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

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

Lex can generate analyzers in either C or Ratfor, a language which can be translated automatically to portable Fortran. It is available on the PDP-11 UNIX, Honeywell GCOS, and IBM OS systems. This manual, however, will only discuss generating analyzers in C on the UNIX system, which is the only supported form of Lex under UNIX Version 7. Lex is designed to simplify interfacing with Yacc, for those with access to this compiler-compiler system.

1. Introduction.

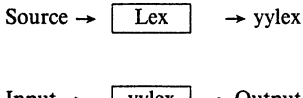
Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified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

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

same and often inappropriate string handling language.

Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called "host languages." Just as general purpose languages can produce code to run on different computer hardware, Lex can write code in different host languages. The host language is used for the output code generated by Lex and also for the program fragments added by the user. Compatible run-time libraries for the different host languages are also provided. This makes Lex adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. At present, the only supported host language is C, although Fortran (in the form of Ratfor [2] has been available in the past. Lex itself exists on UNIX, GCOS, and OS/370; but the code generated by Lex may be taken anywhere the appropriate compilers exist.

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



An overview of Lex
Figure 1

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

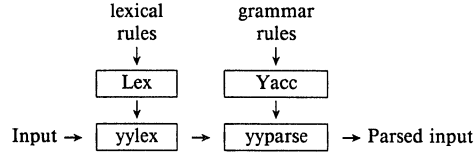
```
%%
[ \t]+$    ;
```

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

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

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

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



Lex with Yacc
Figure 2

Yacc users will realize that the name *yylex* is what Yacc expects its lexical analyzer to be named, so that the use of this name by Lex simplifies interfacing.

Lex generates a deterministic finite automaton from the regular expressions in the source [4]. The automaton is interpreted,

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

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

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

2. Lex Source.

The general format of Lex source is:

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

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

%%

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

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

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

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

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

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

3. Lex Regular Expressions.

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

integer

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

a57D

looks for the string *a57D*.

Operators. The operator characters are

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

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

xyz"++"

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

"xyz++"

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

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

xyz\+\+

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

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

[a–z0–9<>_]

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

[–+0–9]

matches all the digits and the two signs.

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

[^abc]

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

[^a–zA–Z]

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

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

.

is the class of all characters except newline. Escaping into octal is possible although nonportable:

[\40–\176]

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

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

ab?c

matches either *ac* or *abc*.

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

a*

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

a+

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

[a–z]+

is all strings of lower case letters. And

[A–Za–z][A–Za–z0–9]*

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

Alternation and Grouping. The operator | indicates alternation:

(ab | cd)

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

ab | cd

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

(ab | cd+)?(ef)*

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

Context sensitivity. Lex will recognize a small amount of surrounding context. The two simplest operators for this are ^ and $. If the first character of an expression is ^, the expression will only be matched at the beginning of a line (after a newline character, or at the beginning of the input stream). This can never conflict with the other meaning of ^, complementation of character classes, since that only

applies within the [] operators. If the very last character is *$*, the expression will only be matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the / operator character, which indicates trailing context. The expression

ab/cd

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

ab$

is the same as

ab/\n

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

<x>

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

<ONE>

Start conditions are explained more fully later.

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

{digit}

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

a{1,5}

looks for 1 to 5 occurrences of *a*.

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

4. Lex Actions.

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

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

[\t\n] ;

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

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

" "
"\t"
"\n"

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

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

[a–z]+ printf("%s", yytext);

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

[a–z]+ ECHO;

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

Sometimes it is more convenient to know the end of what has been found; hence Lex also provides a count *yyleng* of the number of characters matched. To count both the number of

words and the number of characters in words in the input, the user might write

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

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

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

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

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

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

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

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

```
=-[a-zA-Z]    {
              printf("Op (=-) ambiguous\n");
              yyless(yyleng-1);
              ... action for =- ...
              }
```

which prints a message, returns the letter after the operator to the input stream, and treats the operator as "=-". Alternatively it might be desired to treat this as "= -a". To do this, just return the minus sign as well as the letter to the input:

```
=-[a-zA-Z]    {
              printf("Op (=-) ambiguous\n");
              yyless(yyleng-2);
              ... action for = ...
              }
```

will perform the other interpretation. Note that the expressions for the two cases might more easily be written

```
=-/[A-Za-z]
```

in the first case and

```
=/-[A-Za-z]
```

in the second; no backup would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of "=-3", however, makes

```
=-/[^ \t\n]
```

a still better rule.

In addition to these routines, Lex also permits access to the I/O routines it uses. They are:

1) *input()* which returns the next input character;
2) *output(c)* which writes the character *c* on the output; and
3) *unput(c)* pushes the character *c* back onto the input stream to be read later by *input()*.

By default these routines are provided as macro definitions, but the user can override them and supply private versions. These routines define the relationship between external files and internal characters, and must all be retained or modified consistently. They may be redefined, to cause input or output to be transmitted to or from strange places, including other programs or internal memory; but the character set used must be consistent in all routines; a value of zero returned by *input* must mean end of file; and the relationship between *unput* and *input* must be retained or the Lex lookahead will not work. Lex does not look ahead at all if it does not have to, but every rule ending in + * ? or $ or containing / implies lookahead. Lookahead is also necessary to match an expression that is a prefix of another expression. See below for a discussion

of the character set used by Lex. The standard Lex library imposes a 100 character limit on backup.

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

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

5. Ambiguous Source Rules.

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

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

Thus, suppose the rules

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

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

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

```
'.*'
```

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

'first' quoted string here, 'second' here

the above expression will match

'first' quoted string here, 'second'

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

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

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

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

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

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

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

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

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

Consider the two rules

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

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

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

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

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

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

6. Lex Source Definitions.

Remember the format of the Lex source:

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

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

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

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

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

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

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

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

```
name translation
```

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

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

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

number, a context-sensitive rule such as

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

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

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

7. Usage.

There are two steps in compiling a Lex source program. First, the Lex source must be turned into a generated program in the host general purpose language. Then this program must be compiled and loaded, usually with a library of Lex subroutines. The generated program is on a file named *lex.yy.c*. The I/O library is defined in terms of the C standard library [6].

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

UNIX. The library is accessed by the loader flag *–ll*. So an appropriate set of commands is

```
lex source cc lex.yy.c –ll
```

The resulting program is placed on the usual file *a.out* for later execution. To use Lex with Yacc see below. Although the default Lex I/O routines use the C standard library, the Lex automata themselves do not do so; if private versions of *input, output* and *unput* are given, the library can be avoided.

8. Lex and Yacc.

If you want to use Lex with Yacc, note that what Lex writes is a program named *yylex(),* the name required by Yacc for its analyzer. Normally, the default main program on the Lex library calls this routine, but if Yacc is loaded, and its main program is used, Yacc will call *yylex()*. In this case each Lex rule should end with

```
return(token);
```

where the appropriate token value is returned. An easy way to get access to Yacc's names for tokens is to compile the Lex output file as part of the Yacc output file by placing the line

```
# include "lex.yy.c"
```

in the last section of Yacc input. Supposing the grammar to be named "good" and the lexical rules to be named "better" the UNIX command sequence can just be:

```
yacc good
lex better
cc y.tab.c –ly –ll
```

The Yacc library (–ly) should be loaded before the Lex library, to obtain a main program which invokes the Yacc parser. The generations of Lex and Yacc programs can be done in either order.

9. Examples.

As a trivial problem, consider copying an input file while adding 3 to every positive number divisible by 7. Here is a suitable Lex source program

```
%%
            int k;
[0–9]+      {
            k = atoi(yytext);
            if (k%7 == 0)
                printf("%d", k+3);
            else
                printf("%d",k);
            }
```

to do just that. The rule [0–9]+ recognizes strings of digits; *atoi* converts the digits to binary and stores the result in *k*. The operator % (remainder) is used to check whether *k* is divisible by 7; if it is, it is incremented by 3 as it is written out. It may be objected that this program will alter such input items as *49.63* or *X7*. Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as here:

```
%%
                        int k;
–?[0–9]+                {
                        k = atoi(yytext);
                        printf("%d",
                          k%7 == 0 ? k+3 : k);
                        }
–?[0–9.]+               ECHO;
[A-Za-z][A-Za-z0-9]+    ECHO;
```

Numerical strings containing a "." or preceded by a letter will be picked up by one of the last two rules, and not changed. The *if-else* has been replaced by a C conditional expression to save space; the form *a?b:c* means "if *a* then *b*

else *c*".

For an example of statistics gathering, here is a program which histograms the lengths of words, where a word is defined as a string of letters.

```
        int lengs[100];
%%
[a-z]+  lengs[yyleng]++;
.       |
\n      ;
%%
yywrap()
{
int i;
printf("Length  No. words\n");
for(i=0; i<100; i++)
    if (lengs[i] > 0)
        printf("%5d%10d\n",i,lengs[i]);
return(1);
}
```

This program accumulates the histogram, while producing no output. At the end of the input it prints the table. The final statement *return(1);* indicates that Lex is to perform wrapup. If *yywrap* returns zero (false) it implies that further input is available and the program is to continue reading and processing. To provide a *yywrap* that never returns true causes an infinite loop.

As a larger example, here are some parts of a program written by N. L. Schryer to convert double precision Fortran to single precision Fortran. Because Fortran does not distinguish upper and lower case letters, this routine begins by defining a set of classes including both cases of each letter:

```
a   [aA]
b   [bB]
c   [cC]
...
z   [zZ]
```

An additional class recognizes white space:

```
W   [ \t]*
```

The first rule changes "double precision" to "real", or "DOUBLE PRECISION" to "REAL".

```
{d}{o}{u}{b}{l}{e}{W}{p}{r}{e}{c}{i}{s}{i}{o}{n} {
    printf(yytext[0]=='d'? "real" : "REAL");
    }
```

Care is taken throughout this program to preserve the case (upper or lower) of the original program. The conditional operator is used to select the proper form of the keyword. The next rule copies continuation card indications to avoid confusing them with constants:

```
^"      "[^ 0]      ECHO;
```

In the regular expression, the quotes surround the blanks. It is interpreted as "beginning of line, then five blanks, then anything but blank or zero." Note the two different meanings of ^. There follow some rules to change double precision constants to ordinary floating constants.

```
[0-9]+{W}{d}{W}[+-]?{W}[0-9]+       |
[0-9]+{W}"."{W}{d}{W}[+-]?{W}[0-9]+ |
"."{W}[0-9]+{W}{d}{W}[+-]?{W}[0-9]+ {
    /* convert constants */
    for(p=yytext; *p != 0; p++)
        {
        if (*p == 'd' || *p == 'D')
            *p=+ 'e'-'d';
        ECHO;
        }
```

After the floating point constant is recognized, it is scanned by the *for* loop to find the letter *d* or *D*. The program than adds *'e'-'d'*, which converts it to the next letter of the alphabet. The modified constant, now single-precision, is written out again. There follow a series of names which must be respelled to remove their initial *d*. By using the array *yytext* the same action suffices for all the names (only a sample of a rather long list is given here).

```
{d}{s}{i}{n}        |
{d}{c}{o}{s}        |
{d}{s}{q}{r}{t}     |
{d}{a}{t}{a}{n}     |
...
{d}{f}{l}{o}{a}{t}  printf("%s",yytext+1);
```

Another list of names must have initial *d* changed to initial *a*:

```
{d}{l}{o}{g}        |
{d}{l}{o}{g}10      |
{d}{m}{i}{n}1       |
{d}{m}{a}{x}1       {
                    yytext[0] =+ 'a' - 'd';
                    ECHO;
                    }
```

And one routine must have initial *d* changed to initial *r*:

```
{d}1{m}{a}{c}{h}    {yytext[0] =+ 'r' - 'd';
```

To avoid such names as *dsinx* being detected as instances of *dsin*, some final rules pick up longer words as identifiers and copy some surviving characters:

```
[A-Za-z][A-Za-z0-9]*    |
[0-9]+                  |
```

```
\n                    |
                      ECHO;
```

Note that this program is not complete; it does not deal with the spacing problems in Fortran or with the use of keywords as identifiers.

10. Left Context Sensitivity.

Sometimes it is desirable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The ^ operator, for example, is a prior context operator, recognizing immediately preceding left context just as *$* recognizes immediately following right context. Adjacent left context could be extended, to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often the relevant left context appeared some time earlier, such as at the beginning of a line.

This section describes three means of dealing with different environments: a simple use of flags, when only a few rules change from one environment to another, the use of *start conditions* on rules, and the possibility of making multiple lexical analyzers all run together. In each case, there are rules which recognize the need to change the environment in which the following input text is analyzed, and set some parameter to reflect the change. This may be a flag explicitly tested by the user's action code; such a flag is the simplest way of dealing with the problem, since Lex is not involved at all. It may be more convenient, however, to have Lex remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It will only be recognized when Lex is in that start condition. The current start condition may be changed at any time. Finally, if the sets of rules for the different environments are very dissimilar, clarity may be best achieved by writing several distinct lexical analyzers, and switching from one to another as desired.

Consider the following problem: copy the input to the output, changing the word *magic* to *first* on every line which began with the letter *a*, changing *magic* to *second* on every line which began with the letter *b*, and changing *magic* to *third* on every line which began with the letter *c*. All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a flag:

```
        int flag;
%%
^a      {flag = 'a'; ECHO;}
^b      {flag = 'b'; ECHO;}
^c      {flag = 'c'; ECHO;}
\n      {flag =  0 ; ECHO;}
magic   {
        switch (flag)
        {
        case 'a': printf("first"); break;
        case 'b': printf("second"); break;
        case 'c': printf("third"); break;
        default: ECHO; break;
        }
        }
```

should be adequate.

To handle the same problem with start conditions, each start condition must be introduced to Lex in the definitions section with a line reading

```
%Start    name1 name2 ...
```

where the conditions may be named in any order. The word *Start* may be abbreviated to *s* or *S*. The conditions may be referenced at the head of a rule with the <> brackets:

```
<name1>expression
```

is a rule which is only recognized when Lex is in the start condition *name1*. To enter a start condition, execute the action statement

```
BEGIN name1;
```

which changes the start condition to *name1*. To resume the normal state,

```
BEGIN 0;
```

resets the initial condition of the Lex automaton interpreter. A rule may be active in several start conditions:

```
<name1,name2,name3>
```

is a legal prefix. Any rule not beginning with the <> prefix operator is always active.

The same example as before can be written:

```
%START AA BB CC
%%
^a                  {ECHO; BEGIN AA;}
^b                  {ECHO; BEGIN BB;}
^c                  {ECHO; BEGIN CC;}
\n                  {ECHO; BEGIN 0;}
<AA>magic           printf("first");
<BB>magic           printf("second");
<CC>magic           printf("third");
```

where the logic is exactly the same as in the

previous method of handling the problem, but Lex does the work rather than the user's code.

11. Character Set.

The programs generated by Lex handle character I/O only through the routines *input, output,* and *unput.* Thus the character representation provided in these routines is accepted by Lex and employed to return values in *yytext.* For internal use a character is represented as a small integer which, if the standard library is used, has a value equal to the integer value of the bit pattern representing the character on the host computer. Normally, the letter *a* is represented as the same form as the character constant *'a'*. If this interpretation is changed, by providing I/O routines which translate the characters, Lex must be told about it, by giving a translation table. This table must be in the definitions section, and must be bracketed by lines containing only "%T". The table contains lines of the form

{integer} {character string}

which indicate the value associated with each character. Thus the next example

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

Sample character table.

maps the lower and upper case letters together into the integers 1 through 26, newline into 27, + and – into 28 and 29, and the digits into 30 through 39. Note the escape for newline. If a table is supplied, every character that is to appear either in the rules or in any valid input must be included in the table. No character may be assigned the number 0, and no character may be assigned a bigger number than the size of the hardware character set.

12. Summary of Source Format.

The general form of a Lex source file is:

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

The definitions section contains a combination of

1) Definitions, in the form "name space translation".
2) Included code, in the form "space code".
3) Included code, in the form
   %{
   code
   %}
4) Start conditions, given in the form
   %S name1 name2 ...
5) Character set tables, in the form
   %T
   number space character-string
   ...
   %T
6) Changes to internal array sizes, in the form
   %x nnn
   where *nnn* is a decimal integer representing an array size and *x* selects the parameter as follows:

Letter	Parameter
p	positions
n	states
e	tree nodes
a	transitions
k	packed character classes
o	output array size

Lines in the rules section have the form "expression action" where the action may be continued on succeeding lines by using braces to delimit it.

Regular expressions in Lex use the following operators:

x	the character "x"
"x"	an "x", even if x is an operator.
\x	an "x", even if x is an operator.
[xy]	the character x or y.
[x–z]	the characters x, y or z.
[^x]	any character but x.
.	any character but newline.
^x	an x at the beginning of a line.
<y>x	an x when Lex is in start condition y.
x$	an x at the end of a line.
x?	an optional x.

x*	0,1,2, ... instances of x.
x+	1,2,3, ... instances of x.
x|y	an x or a y.
(x)	an x.
x/y	an x but only if followed by y.
{xx}	the translation of xx from the definitions section.
x{m,n}	*m* through *n* occurrences of x

13. Caveats and Bugs.

There are pathological expressions which produce exponential growth of the tables when converted to deterministic machines; fortunately, they are rare.

REJECT does not rescan the input; instead it remembers the results of the previous scan. This means that if a rule with trailing context is found, and REJECT executed, the user must not have used *unput* to change the characters forthcoming from the input stream. This is the only restriction on the user's ability to manipulate the not-yet-processed input.

14. Acknowledgments.

As should be obvious from the above, the outside of Lex is patterned on Yacc and the inside on Aho's string matching routines. Therefore, both S. C. Johnson and A. V. Aho are really originators of much of Lex, as well as debuggers of it. Many thanks are due to both.

The code of the current version of Lex was designed, written, and debugged by Eric Schmidt.

15. References.

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, N. J. (1978).
2. B. W. Kernighan, *Ratfor: A Preprocessor for a Rational Fortran,* Software – Practice and Experience, **5**, pp. 395-496 (1975).
3. S. C. Johnson, *Yacc: Yet Another Compiler Compiler,* Computing Science Technical Report No. 32, 1975, Bell Laboratories, Murray Hill, NJ 07974.
4. A. V. Aho and M. J. Corasick, *Efficient String Matching: An Aid to Bibliographic Search,* Comm. ACM **18,** 333-340 (1975).
5. B. W. Kernighan, D. M. Ritchie and K. L. Thompson, *QED Text Editor,* Computing Science Technical Report No. 5, 1972, Bell Laboratories, Murray Hill, NJ 07974.
6. D. M. Ritchie, private communication. See also M. E. Lesk, *The Portable C Library,* Computing Science Technical Report No. 31, Bell Laboratories, Murray Hill, NJ 07974.

The M4 Macro Processor

Brian W. Kernighan

Dennis M. Ritchie

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

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

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

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

This paper is a user's manual for M4.

Introduction

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

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

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

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

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

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.

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.

PS 18
1

Screen Updating and Cursor Movement Optimization: A Library Package

Kenneth C. R. C. Arnold

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

ABSTRACT

This document describes a package of C library functions which allow the user to:

- update a screen with reasonable optimization,
- get input from the terminal in a screen-oriented fashion, and
- independent from the above, move the cursor optimally from one point to another.

These routines all use the **termcap**(5) database to describe the capabilities of the terminal.

Acknowledgements

This package would not exist without the work of Bill Joy, who, in writing his editor, created the capability to generally describe terminals, wrote the routines which read this database, and, most importantly, those which implement optimal cursor movement, which routines I have simply lifted nearly intact. Doug Merritt and Kurt Shoens also were extremely important, as were both willing to waste time listening to me rant and rave. The help and/or support of Ken Abrams, Alan Char, Mark Horton, and Joe Kalash, was, and is, also greatly appreciated.

Revised 16 April 1986

Contents

Appendixes

The M4 Macro Processor

Brian W. Kernighan

Dennis M. Ritchie

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

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

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

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

This paper is a user's manual for M4.

Introduction

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

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

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

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

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

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.

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.

Screen Updating and Cursor Movement Optimization: A Library Package

Kenneth C. R. C. Arnold

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

ABSTRACT

This document describes a package of C library functions which allow the user to:

- update a screen with reasonable optimization,
- get input from the terminal in a screen-oriented fashion, and
- independent from the above, move the cursor optimally from one point to another.

These routines all use the **termcap**(5) database to describe the capabilities of the terminal.

Acknowledgements

This package would not exist without the work of Bill Joy, who, in writing his editor, created the capability to generally describe terminals, wrote the routines which read this database, and, most importantly, those which implement optimal cursor movement, which routines I have simply lifted nearly intact. Doug Merritt and Kurt Shoens also were extremely important, as were both willing to waste time listening to me rant and rave. The help and/or support of Ken Abrams, Alan Char, Mark Horton, and Joe Kalash, was, and is, also greatly appreciated.

Revised 16 April 1986

Contents

Appendixes

1. Overview

In making available the generalized terminal descriptions in **termcap**(5), much information was made available to the programmer, but little work was taken out of one's hands. The purpose of this package is to allow the C programmer to do the most common type of terminal dependent functions, those of movement optimization and optimal screen updating, without doing any of the dirty work, and (hopefully) with nearly as much ease as is necessary to simply print or read things.

The package is split into three parts: (1) Screen updating; (2) Screen updating with user input; and (3) Cursor motion optimization.

It is possible to use the motion optimization without using either of the other two, and screen updating and input can be done without any programmer knowledge of the motion optimization, or indeed the database itself.

1.1. Terminology (or, Words You Can Say to Sound Brilliant)

In this document, the following terminology is kept to with reasonable consistency:

window: An internal representation containing an image of what a section of the terminal screen may look like at some point in time. This subsection can either encompass the entire terminal screen, or any smaller portion down to a single character within that screen.

terminal: Sometimes called ***terminal screen***. The package's idea of what the terminal's screen currently looks like, *i.e.*, what the user sees now. This is a special *screen*:

screen: This is a subset of windows which are as large as the terminal screen, *i.e.*, they start at the upper left hand corner and encompass the lower right hand corner. One of these, *stdscr*, is automatically provided for the programmer.

1.2. Compiling Things

In order to use the library, it is necessary to have certain types and variables defined. Therefore, the programmer must have a line:

#include <curses.h>

at the top of the program source. The header file **<curses.h>** needs to include **<sgtty.h>**, so the one should not do so oneself[1]. Also, compilations should have the following form:

cc [*flags*] file ... **-lcurses -ltermcap**

1.3. Screen Updating

In order to update the screen optimally, it is necessary for the routines to know what the screen currently looks like and what the programmer wants it to look like next. For this purpose, a data type (structure) named *WINDOW* is defined which describes a window image to the routines, including its starting position on the screen (the (y, x) co-ordinates of the upper left hand corner) and its size. One of these (called *curscr* for *current screen*) is a screen image of what the terminal currently looks like. Another screen (called *stdscr*, for *standard screen*) is provided by default to make changes on.

---

[1] The screen package also uses the Standard I/O library, so **<curses.h>** includes **<stdio.h>**. It is redundant (but harmless) for the programmer to do it, too.

A window is a purely internal representation. It is used to build and store a potential image of a portion of the terminal. It doesn't bear any necessary relation to what is really on the terminal screen. It is more like an array of characters on which to make changes.

When one has a window which describes what some part the terminal should look like, the routine *refresh()* (or *wrefresh()* if the window is not *stdscr*) is called. *refresh()* makes the terminal, in the area covered by the window, look like that window. Note, therefore, that changing something on a window *does **not** change the terminal.* Actual updates to the terminal screen are made only by calling *refresh()* or *wrefresh()*. This allows the programmer to maintain several different ideas of what a portion of the terminal screen should look like. Also, changes can be made to windows in any order, without regard to motion efficiency. Then, at will, the programmer can effectively say "make it look like this", and let the package worry about the best way to do this.

1.4. Naming Conventions

As hinted above, the routines can use several windows, but two are automatically given: *curscr*, which knows what the terminal looks like, and *stdscr*, which is what the programmer wants the terminal to look like next. The user should never really access *curscr* directly. Changes should be made to the appropriate screen, and then the routine *refresh()* (or *wrefresh()*) should be called.

Many functions are set up to deal with *stdscr* as a default screen. For example, to add a character to *stdscr*, one calls *addch()* with the desired character. If a different window is to be used, the routine *waddch()* (for window-specific *addch()*) is provided[2]. This convention of prepending function names with a "**w**" when they are to be applied to specific windows is consistent. The only routines which do *not* do this are those to which a window must always be specified.

In order to move the current (y, x) co-ordinates from one point to another, the routines *move()* and *wmove()* are provided. However, it is often desirable to first move and then perform some I/O operation. In order to avoid clumsyness, most I/O routines can be preceded by the prefix "**mv**" and the desired (y, x) co-ordinates then can be added to the arguments to the function. For example, the calls

```
move(y, x);
addch(ch);
```

can be replaced by

```
mvaddch(y, x, ch);
```

and

```
wmove(win, y, x);
waddch(win, ch);
```

can be replaced by

```
mvwaddch(win, y, x, ch);
```

Note that the window description pointer (*win*) comes before the added (y, x) co-ordinates. If such pointers are need, they are always the first parameters passed.

---

[2] Actually, *addch()* is really a "#define" macro with arguments, as are most of the "functions" which deal with *stdscr* as a default.

2. Variables

Many variables which are used to describe the terminal environment are available to the programmer. They are:

type	name	description
WINDOW *	curscr	current version of the screen (terminal screen).
WINDOW *	stdscr	standard screen. Most updates are usually done here.
char *	Def_term	default terminal type if type cannot be determined
bool	My_term	use the terminal specification in *Def_term* as terminal, irrelevant of real terminal type
char *	ttytype	full name of the current terminal.
int	LINES	number of lines on the terminal
int	COLS	number of columns on the terminal
int	ERR	error flag returned by routines on a fail.
int	OK	error flag returned by routines when things go right.

There are also several "#define" constants and types which are of general usefulness:

reg	storage class "register" (*e.g., reg int i;*)
bool	boolean type, actually a "char" (*e.g., bool doneit;*)
TRUE	boolean "true" flag (1).
FALSE	boolean "false" flag (0).

3. Usage

This is a description of how to actually use the screen package. In it, we assume all updating, reading, etc. is applied to *stdscr*. All instructions will work on any window, with changing the function name and parameters as mentioned above.

3.1. Starting up

In order to use the screen package, the routines must know about terminal characteristics, and the space for *curscr* and *stdscr* must be allocated. These functions are performed by *initscr()*. Since it must allocate space for the windows, it can overflow core when attempting to do so. On this rather rare occasion, *initscr()* returns ERR. *initscr()* must ***always*** be called before any of the routines which affect windows are used. If it is not, the program will core dump as soon as either *curscr* or *stdscr* are referenced. However, it is usually best to wait to call it until after you are sure you will need it, like after checking for startup errors. Terminal status changing routines like *nl()* and *cbreak()* should be called after *initscr()*.

Now that the screen windows have been allocated, you can set them up for the run. If you want to, say, allow the window to scroll, use *scrollok()*. If you want the cursor to be left after the last change, use *leaveok()*. If this isn't done, *refresh()* will move the cursor to the window's current (y, x) co-ordinates after updating it. New windows of your own can be created, too, by using the functions *newwin()* and *subwin()*. *delwin()* will allow you to get rid of old windows. If you wish to change the official size of the terminal by hand, just set the variables *LINES* and *COLS* to be what you want, and then call *initscr()*. This is best done before, but can be done either before or after, the first call to *initscr()*, as it will always delete any existing *stdscr* and/or *curscr* before creating new ones.

3.2. The Nitty-Gritty

3.2.1. Output

Now that we have set things up, we will want to actually update the terminal. The basic functions used to change what will go on a window are *addch()* and *move()*. *addch()* adds a

character at the current (y, x) co-ordinates, returning ERR if it would cause the window to illegally scroll, *i.e.*, printing a character in the lower right-hand corner of a terminal which automatically scrolls if scrolling is not allowed. *move()* changes the current (y, x) co-ordinates to whatever you want them to be. It returns ERR if you try to move off the window when scrolling is not allowed. As mentioned above, you can combine the two into *mvaddch()* to do both things in one fell swoop.

The other output functions, such as *addstr()* and *printw()*, all call *addch()* to add characters to the window.

After you have put on the window what you want there, when you want the portion of the terminal covered by the window to be made to look like it, you must call *refresh()*. In order to optimize finding changes, *refresh()* assumes that any part of the window not changed since the last *refresh()* of that window has not been changed on the terminal, *i.e.*, that you have not refreshed a portion of the terminal with an overlapping window. If this is not the case, the routines *touchwin()*, *touchline()*, and *touchoverlap()* are provided to make it look like a desired part of window has been changed, thus forcing *refresh()* check that whole subsection of the terminal for changes.

If you call *wrefresh()* with *curscr*, it will make the screen look like *curscr* thinks it looks like. This is useful for implementing a command which would redraw the screen in case it get messed up.

3.2.2. Input

Input is essentially a mirror image of output. The complementary function to *addch()* is *getch()* which, if echo is set, will call *addch()* to echo the character. Since the screen package needs to know what is on the terminal at all times, if characters are to be echoed, the tty must be in raw or cbreak mode. If it is not, *getch()* sets it to be cbreak, and then reads in the character.

3.2.3. Miscellaneous

All sorts of fun functions exists for maintaining and changing information about the windows. For the most part, the descriptions in section 5.4. should suffice.

3.3. Finishing up

In order to do certain optimizations, and, on some terminals, to work at all, some things must be done before the screen routines start up. These functions are performed in *getttmode()* and *setterm()*, which are called by *initscr()*. In order to clean up after the routines, the routine *endwin()* is provided. It restores tty modes to what they were when *initscr()* was first called. Thus, anytime after the call to initscr, *endwin()* should be called before exiting.

4. Cursor Motion Optimization: Standing Alone

It is possible to use the cursor optimization functions of this screen package without the overhead and additional size of the screen updating functions. The screen updating functions are designed for uses where parts of the screen are changed, but the overall image remains the same. This includes such programs as **rogue** and **vi**[3]. Certain other programs will find it difficult to use these functions in this manner without considerable unnecessary program overhead. For such applications, such as some "*crt hacks*"[4] and optimizing **more**(1)-type pro-

[3] **rogue** actually uses these functions, **vi** does not.

[4] Graphics programs designed to run on character-oriented terminals. I could name many, but they come and go, so the list would be quickly out of date. Recently, there have been programs such as **rain**, **rocket**, and **gun**.

grams, all that is needed is the motion optimizations. This, therefore, is a description of what some of what goes on at the lower levels of this screen package. The descriptions assume a certain amount of familiarity with programming problems and some finer points of C. None of it is terribly difficult, but you should be forewarned.

4.1. Terminal Information

In order to use a terminal's features to the best of a program's abilities, it must first know what they are[5]. The **termcap**(5) database describes these, but a certain amount of decoding is necessary, and there are, of course, both efficient and inefficient ways of reading them in. The algorithm that the uses is taken from **vi** and is hideously efficient. It reads them in a tight loop into a set of variables whose names are two uppercase letters with some mnemonic value. For example, *HO* is a string which moves the cursor to the "home" position[6]. As there are two types of variables involving ttys, there are two routines. The first, *gettmode()*, sets some variables based upon the tty modes accessed by **gtty**(2) and **stty**(2) The second, *setterm()*, a larger task by reading in the descriptions from the **termcap**(5) database. This is the way these routines are used by *initscr()*:

```
if (isatty(0)) {
        gettmode();
        if ((sp=getenv("TERM")) != NULL)
                setterm(sp);
                else
                        setterm(Def_term);
}
else
        setterm(Def_term);
_puts(TI);
_puts(VS);
```

isatty() checks to see if file descriptor 0 is a terminal[7]. If it is, *gettmode()* sets the terminal description modes from a **gtty**(2) *getenv()* is then called to get the name of the terminal, and that value (if there is one) is passed to *setterm()*, which reads in the variables from **termcap**(5) associated with that terminal. (*getenv()* returns a pointer to a string containing the name of the terminal, which we save in the character pointer *sp*.) If *isatty()* returns false, the default terminal *Def_term* is used. The *TI* and *VS* sequences initialize the terminal (*_puts()* is a macro which uses *tputs()* (see **termcap**(3)) and *_putchar()* to put out a string). *endwin()* undoes these things.

4.2. Movement Optimizations, or, Getting Over Yonder

Now that we have all this useful information, it would be nice to do something with it[8]. The most difficult thing to do properly is motion optimization. When you consider how many different features various terminals have (tabs, backtabs, non-destructive space, home

[5] If this comes as any surprise to you, there's this tower in Paris they're thinking of junking that I can let you have for a song.

[6] These names are identical to those variables used in the **termcap**(5) database to describe each capability. See Appendix A for a complete list of those read, and the **termcap**(5) manual page for a full description.

[7] *isatty()* is defined in the default C library function routines. It does a **gtty**(2) on the descriptor and checks the return value.

[8] Actually, it *can* be emotionally fulfilling just to get the information. This is usually only true, however, if you have the social life of a kumquat.

sequences, absolute tabs, ...) you can see that deciding how to get from here to there can be a decidedly non-trivial task. The editor **vi** uses many of these features, and the routines it uses to do this take up many pages of code. Fortunately, I was able to liberate them with the author's permission, and use them here.

After using *gettmode()* and *setterm()* to get the terminal descriptions, the function *mvcur()* deals with this task. It usage is simple: you simply tell it where you are now and where you want to go. For example

```
mvcur(0, 0, LINES/2, COLS/2)
```

would move the cursor from the home position (0, 0) to the middle of the screen. If you wish to force absolute addressing, you can use the function *tgoto()* from the **termlib**(7) routines, or you can tell *mvcur()* that you are impossibly far away, like Cleveland. For example, to absolutely address the lower left hand corner of the screen from anywhere just claim that you are in the upper right hand corner:

```
mvcur(0, COLS-1, LINES-1, 0)
```

5. The Functions

In the following definitions, "†" means that the "function" is really a "#define" macro with arguments. This means that it will not show up in stack traces in the debugger, or, in the case of such functions as *addch()*, it will show up as it's "w" counterpart. The arguments are given to show the order and type of each. Their names are not mandatory, just suggestive.

5.1. Output Functions

addch(ch) †
char ch;

waddch(win, ch)
*WINDOW *win;*
char ch;

Add the character *ch* on the window at the current (y, x) co-ordinates. If the character is a newline ('\n') the line will be cleared to the end, and the current (y, x) co-ordinates will be changed to the beginning off the next line if newline mapping is on, or to the next line at the same x co-ordinate if it is off. A return ('\r') will move to the beginning of the line on the window. Tabs ('\t') will be expanded into spaces in the normal tabstop positions of every eight characters. This returns ERR if it would cause the screen to scroll illegally.

addstr(str) †
*char *str;*

waddstr(win, str)
*WINDOW *win;*
*char *str;*

Add the string pointed to by *str* on the window at the current (y, x) co-ordinates. This returns ERR if it would cause the screen to scroll illegally. In this case, it will put on as much as it can.

box(win, vert, hor)
WINDOW **win;*
char *vert, hor;*

Draws a box around the window using *vert* as the character for drawing the vertical sides, and *hor* for drawing the horizontal lines. If scrolling is not allowed, and the window encompasses the lower right-hand corner of the terminal, the corners are left blank to avoid a scroll.

clear() †

wclear(win)
WINDOW **win;*

Resets the entire window to blanks. If *win* is a screen, this sets the clear flag, which will cause a clear-screen sequence to be sent on the next *refresh()* call. This also moves the current (y, x) co-ordinates to (0, 0).

clearok(scr, boolf) †
WINDOW **scr;*
bool *boolf;*

Sets the clear flag for the screen *scr*. If *boolf* is TRUE, this will force a clear-screen to be printed on the next *refresh()*, or stop it from doing so if *boolf* is FALSE. This only works on screens, and, unlike *clear()*, does not alter the contents of the screen. If *scr* is *curscr*, the next *refresh()* call will cause a clear-screen, even if the window passed to *refresh()* is not a screen.

clrtobot() †

wclrtobot(win)
WINDOW **win;*

Wipes the window clear from the current (y, x) co-ordinates to the bottom. This does not force a clear-screen sequence on the next refresh under any circumstances. This has no associated "**mv**" command.

clrtoeol() †

wclrtoeol(win)
WINDOW **win;*

Wipes the window clear from the current (y, x) co-ordinates to the end of the line. This has no associated "**mv**" command.

delch()

wdelch(win)

*WINDOW *win;*

Delete the character at the current (y, x) co-ordinates. Each character after it on the line shifts to the left, and the last character becomes blank.

deleteln()

wdeleteln(win)
*WINDOW *win;*

Delete the current line. Every line below the current one will move up, and the bottom line will become blank. The current (y, x) co-ordinates will remain unchanged.

erase() †

werase(win)
*WINDOW *win;*

Erases the window to blanks without setting the clear flag. This is analagous to *clear()*, except that it never causes a clear-screen sequence to be generated on a *refresh()*. This has no associated "**mv**" command.

flushok(win, boolf) †
*WINDOW *win;*
bool boolf;

Normally, *refresh() fflush()*'s *stdout* when it is finished. *flushok()* allows you to control this. if *boolf* is TRUE (*i.e.*, non-zero) it will do the *fflush()*; if it is FALSE. it will not.

idlok(win, boolf)
*WINDOW *win;*
bool boolf;

Reserved for future use. This will eventually signal to *refresh()* that it is all right to use the insert and delete line sequences when updating the window.

insch(c)
char c;

winsch(win, c)
*WINDOW *win;*
char c;

Insert *c* at the current (y, x) co-ordinates Each character after it shifts to the right, and the last character disappears. This returns ERR if it would cause the screen to scroll illegally.

insertln()

winsertln(win)
*WINDOW *win;*

Insert a line above the current one. Every line below the current line will be shifted down, and the bottom line will disappear. The current line will become blank, and the current (y, x) co-ordinates will remain unchanged.

move(y, x) †
int y, x;

wmove(win, y, x)
*WINDOW *win;*
int y, x;

Change the current (y, x) co-ordinates of the window to (*y, x*). This returns ERR if it would cause the screen to scroll illegally.

overlay(win1, win2)
*WINDOW *win1, *win2;*

Overlay *win1* on *win2*. The contents of *win1*, insofar as they fit, are placed on *win2* at their starting (y, x) co-ordinates. This is done non-destructively, i.e., blanks on *win1* leave the contents of the space on *win2* untouched.

overwrite(win1, win2)
*WINDOW *win1, *win2;*

Overwrite *win1* on *win2*. The contents of *win1*, insofar as they fit, are placed on *win2* at their starting (y, x) co-ordinates. This is done destructively, *i.e.*, blanks on *win1* become blank on *win2*.

printw(fmt, arg1, arg2, ...)
*char *fmt;*

wprintw(win, fmt, arg1, arg2, ...)
*WINDOW *win;*
*char *fmt;*

Performs a *printf()* on the window starting at the current (y, x) co-ordinates. It uses *addstr()* to add the string on the window. It is often advisable to use the field width options of *printf()* to avoid leaving things on the window from earlier calls. This returns ERR if it would cause the screen to scroll illegally.

refresh() †

wrefresh(win)
*WINDOW *win;*

Synchronize the terminal screen with the desired window. If the window is not a screen, only that part covered by it is updated. This returns ERR if it would cause the screen to

scroll illegally. In this case, it will update whatever it can without causing the scroll.

As a special case, if *wrefresh()* is called with the window *curscr* the screen is cleared and repainted as it is currently. This is very useful for allowing the redrawing of the screen when the user has garbage dumped on his terminal.

standout() †

wstandout(win)
*WINDOW *win;*

standend() †

wstandend(win)
*WINDOW *win;*

Start and stop putting characters onto *win* in standout mode. *standout()* causes any characters added to the window to be put in standout mode on the terminal (if it has that capability). *standend()* stops this. The sequences *SO* and *SE* (or *US* and *UE* if they are not defined) are used (see Appendix A).

5.2. Input Functions

cbreak() †

nocbreak() †

crmode() †

nocrmode() †

Set or unset the terminal to/from cbreak mode. The misnamed macros *crmode()* and *nocrmode()* are retained for backwards compatibility with ealier versions of the library.

echo() †

noecho() †

Sets the terminal to echo or not echo characters.

getch() †

wgetch(win)
*WINDOW *win;*

Gets a character from the terminal and (if necessary) echos it on the window. This returns ERR if it would cause the screen to scroll illegally. Otherwise, the character gotten is returned. If *noecho* has been set, then the window is left unaltered. In order to retain control of the terminal, it is necessary to have one of *noecho*, *cbreak*, or *rawmode* set. If you do not set one, whatever routine you call to read characters will set *cbreak* for you,

and then reset to the original mode when finished.

getstr(str) †
char **str;*

wgetstr(win, str)
WINDOW **win;*
char **str;*

Get a string through the window and put it in the location pointed to by *str*, which is assumed to be large enough to handle it. It sets tty modes if necessary, and then calls *getch()* (or *wgetch(win)*) to get the characters needed to fill in the string until a newline or EOF is encountered. The newline stripped off the string. This returns ERR if it would cause the screen to scroll illegally.

_putchar(c)
char *c;*

Put out a character using the *putchar()* macro. This function is used to output every character that **curses** generates. Thus, it can be redefined by the user who wants to do non-standard things with the output. It is named with an initial "_" because it usually should be invisible to the programmer.

raw() †

noraw() †

Set or unset the terminal to/from raw mode. On version 7 UNIX[9] this also turns of newline mapping (see *nl()*).

scanw(fmt, arg1, arg2, ...)
char **fmt;*

wscanw(win, fmt, arg1, arg2, ...)
WINDOW **win;*
char **fmt;*

Perform a *scanf()* through the window using *fmt*. It does this using consecutive *getch()*'s (or *wgetch(win)*'s). This returns ERR if it would cause the screen to scroll illegally.

5.3. Miscellaneous Functions

baudrate() †

Returns the output baud rate of the terminal. This is a system dependent constant (defined in **<sys/tty.h>** on BSD systems, which is included by **<curses.h>**).

[9] UNIX is a trademark of Bell Laboratories.

delwin(win)
*WINDOW *win;*

Deletes the window from existence. All resources are freed for future use by **calloc**(3). If a window has a *subwin()* allocated window inside of it, deleting the outer window the subwindow is not affected, even though this does invalidate it. Therefore, subwindows should be deleted before their outer windows are.

endwin()

Finish up window routines before exit. This restores the terminal to the state it was before *initscr()* (or *gettmode()* and *setterm()*) was called. It should always be called before exiting. It does not exit. This is especially useful for resetting tty stats when trapping rubouts via **signal**(2).

erasechar() †

Returns the erase character for the terminal, *i.e.*, the character used by the user to erase a single character from the input.

*char **
getcap(str)
*char *str;*

Return a pointer to the **termcap** capability described by *str* (see **termcap**(5) for details).

getyx(win, y, x) †
*WINDOW *win;*
int y, x;

Puts the current (y, x) co-ordinates of *win* in the variables *y* and *x*. Since it is a macro, not a function, you do not pass the address of *y* and *x*.

inch() †

winch(win) †
*WINDOW *win;*

Returns the character at the current on the given window. This does not make any changes to the window.

initscr()

Initialize the screen routines. This must be called before any of the screen routines are used. It initializes the terminal-type data and such, and without it none of the routines can operate. If standard input is not a tty, it sets the specifications to the terminal whose name is pointed to by *Def_term* (initialy "dumb"). If the boolean *My_term* is true, *Def_term* is always used. If the system supports the **TIOCGWINSZ** *ioctl(2)* call, it is used to get the number of lines and columns for the terminal, otherwise it is taken

from the **termcap** description.

killchar() †

Returns the line kill character for the terminal, *i.e.*, the character used by the user to erase an entire line from the input.

leaveok(win, boolf) †
WINDOW **win;*
bool *boolf;*

Sets the boolean flag for leaving the cursor after the last change. If *boolf* is TRUE, the cursor will be left after the last update on the terminal, and the current (y, x) co-ordinates for *win* will be changed accordingly. If it is FALSE, it will be moved to the current (y, x) co-ordinates. This flag (initialy FALSE) retains its value until changed by the user.

longname(termbuf, name)
char **termbuf, *name;*

fullname(termbuf, name)
char **termbuf, *name;*

longname() fills in *name* with the long name of the terminal described by the **termcap** entry in *termbuf*. It is generally of little use, but is nice for telling the user in a readable format what terminal we think he has. This is available in the global variable *ttytype*. *termbuf* is usually set via the termlib routine *tgetent()*. *fullname()* is the same as *longname()*, except that it gives the fullest name given in the entry, which can be quite verbose.

mvwin(win, y, x)
WINDOW **win;*
int *y, x;*

Move the home position of the window *win* from its current starting coordinates to (*y, x*). If that would put part or all of the window off the edge of the terminal screen, *mvwin()* returns ERR and does not change anything. For subwindows, *mvwin()* also returns ERR if you attempt to move it off its main window. If you move a main window, all subwindows are moved along with it.

WINDOW *
newwin(lines, cols, begin_y, begin_x)
int *lines, cols, begin_y, begin_x;*

Create a new window with *lines* lines and *cols* columns starting at position (*begin_y, begin_x*). If either *lines* or *cols* is 0 (zero), that dimension will be set to (*LINES* – *begin_y*) or (*COLS* – *begin_x*) respectively. Thus, to get a new window of dimensions *LINES* × *COLS*, use *newwin(0, 0, 0, 0)*.

nl() †

nonl() †

Set or unset the terminal to/from nl mode, *i.e.*, start/stop the system from mapping <RETURN> to <LINE-FEED>. If the mapping is not done, *refresh()* can do more optimization, so it is recommended, but not required, to turn it off.

scrollok(win, boolf) †
WINDOW **win;*
bool *boolf;*

Set the scroll flag for the given window. If *boolf* is FALSE, scrolling is not allowed. This is its default setting.

touchline(win, y, startx, endx)
WINDOW **win;*
int *y, startx, endx;*

This function performs a function similar to *touchwin()* on a single line. It marks the first change for the given line to be *startx*, if it is before the current first change mark, and the last change mark is set to be *endx* if it is currently less than *endx*.

touchoverlap(win1, win2)
WINDOW **win1, *win2;*

Touch the window *win2* in the area which overlaps with *win1*. If they do not overlap, no changes are made.

touchwin(win)
WINDOW **win;*

Make it appear that the every location on the window has been changed. This is usually only needed for refreshes with overlapping windows.

WINDOW *
subwin(win, lines, cols, begin_y, begin_x)
WINDOW **win;*
int *lines, cols, begin_y, begin_x;*

Create a new window with *lines* lines and *cols* columns starting at position (*begin_y, begin_x*) inside the window *win*. This means that any change made to either window in the area covered by the subwindow will be made on both windows. *begin_y, begin_x* are specified relative to the overall screen, not the relative (0, 0) of *win*. If either *lines* or *cols* is 0 (zero), that dimension will be set to (*LINES* - *begin_y*) or (*COLS* - *begin_x*) respectively.

unctrl(ch) †
char *ch;*

This is actually a debug function for the library, but it is of general usefulness. It returns a string which is a representation of *ch*. Control characters become their uppercase equivalents preceded by a "^". Other letters stay just as they are. To use *unctrl()*, you may have to have **#include <unctrl.h>** in your file.

5.4. Details

gettmode()

Get the tty stats. This is normally called by *initscr()*.

mvcur(lasty, lastx, newy, newx)
int lasty, lastx, newy, newx;

Moves the terminal's cursor from (*lasty, lastx*) to (*newy, newx*) in an approximation of optimal fashion. This routine uses the functions borrowed from *ex* version 2.6. It is possible to use this optimization without the benefit of the screen routines. With the screen routines, this should not be called by the user. *move()* and *refresh()* should be used to move the cursor position, so that the routines know what's going on.

scroll(win)
*WINDOW *win;*

Scroll the window upward one line. This is normally not used by the user.

savetty() †

resetty() †

savetty() saves the current tty characteristic flags. *resetty()* restores them to what *savetty()* stored. These functions are performed automatically by *initscr()* and *endwin()*.

setterm(name)
*char *name;*

Set the terminal characteristics to be those of the terminal named *name*, getting the terminal size from the **TIOCGWINSZ** *ioctl(2)* if it exists, otherwise from the environment. This is normally called by *initscr()*.

tstp()

If the new **tty**(4) driver is in use, this function will save the current tty state and then put the process to sleep. When the process gets restarted, it restores the tty state and then calls *wrefresh(curscr)* to redraw the screen. *initscr()* sets the signal SIGTSTP to trap to this routine.

1. Capabilities from termcap

1.1. Disclaimer

The description of terminals is a difficult business, and we only attempt to summarize the capabilities here: for a full description see **termcap**(5).

1.2. Overview

Capabilities from **termcap** are of three kinds: string valued options, numeric valued options, and boolean options. The string valued options are the most complicated, since they may include padding information, which we describe now.

Intelligent terminals often require padding on intelligent operations at high (and sometimes even low) speed. This is specified by a number before the string in the capability, and has meaning for the capabilities which have a **P** at the front of their comment. This normally is a number of milliseconds to pad the operation. In the current system which has no true programmable delays, we do this by sending a sequence of pad characters (normally nulls, but can be changed (specified by *PC*)). In some cases, the pad is better computed as some number of milliseconds times the number of affected lines (to the bottom of the screen usually, except when terminals have insert modes which will shift several lines.) This is specified as, i e.g. , **12***. before the capability, to say 12 milliseconds per affected whatever (currently always line). Capabilities where this makes sense say **P***.

1.3. Variables Set By setterm()

variables set by *setterm()*

Type	Name	Pad	Description
char *	AL	P*	Add new blank Line
bool	AM		Automatic Margins
char *	BC		Back Cursor movement
bool	BS		BackSpace works
char *	BT	P	Back Tab
bool	CA		Cursor Addressable
char *	CD	P*	Clear to end of Display
char *	CE	P	Clear to End of line
char *	CL	P*	CLear screen
char *	CM	P	Cursor Motion
char *	DC	P*	Delete Character
char *	DL	P*	Delete Line sequence
char *	DM		Delete Mode (enter)
char *	DO		DOwn line sequence
char *	ED		End Delete mode
bool	EO		can Erase Overstrikes with ''
char *	EI		End Insert mode
char *	HO		HOme cursor
bool	HZ		HaZeltine ~ braindamage
char *	IC	P	Insert Character
bool	IN		Insert-Null blessing
char *	IM		enter Insert Mode (IC usually set, too)
char *	IP	P*	Pad after char Inserted using IM+IE

variables set by *setterm()*

Type	Name	Pad	Description
char *	LL		quick to Last Line, column 0
char *	MA		ctrl character MAp for cmd mode
bool	MI		can Move in Insert mode
bool	NC		No Cr: \r sends \r\n then eats \n
char *	ND		Non-Destructive space
bool	OS		OverStrike works
char	PC		Pad Character
char *	SE		Standout End (may leave space)
char *	SF	P	Scroll Forwards
char *	SO		Stand Out begin (may leave space)
char *	SR	P	Scroll in Reverse
char *	TA	P	TAb (not ^I or with padding)
char *	TE		Terminal address enable Ending sequence
char *	TI		Terminal address enable Initialization
char *	UC		Underline a single Character
char *	UE		Underline Ending sequence
bool	UL		UnderLining works even though !OS
char *	UP		UPline
char *	US		Underline Starting sequence
char *	VB		Visible Bell
char *	VE		Visual End sequence
char *	VS		Visual Start sequence
bool	XN		a Newline gets eaten after wrap

Names starting with ***X*** are reserved for severely nauseous glitches

For purposes of *standout()*, if *SG()* is not 0, *SO()* is set to *NULL()*, and if *UG()* is not *0()*, *US()* is set to *NULL()*. If, after this, *SO()* is *NULL()*, and *US()* is not, *SO()* is set to be *US()*, and *SE()* is set to be *UE()*.

1.4. Variables Set By gettmode()

variables set by *gettmode()*

type	name	description
bool	NONL	Term can't hack linefeeds doing a CR
bool	GT	Gtty indicates Tabs
bool	UPPERCASE	Terminal generates only uppercase letters

1. The WINDOW structure

The WINDOW structure is defined as follows:

```
/*
 * Copyright (c) 1980 Regents of the University of California.
 * All rights reserved.  The Berkeley software License Agreement
 * specifies the terms and conditions for redistribution.
 *
 *      @(#)win_st.c    6.1 (Berkeley) 4/24/86";
 */

# define        WINDOW struct _win_st

struct _win_st {
        short           _cury, _curx;
        short           _maxy, _maxx;
        short           _begy, _begx;
        short           _flags;
        short           _ch_off;
        bool            _clear;
        bool            _leave;
        bool            _scroll;
        char            **_y;
        short           *_firstch;
        short           *_lastch;
        struct _win_st  *_nextp, *_orig;
};

# define        _ENDLINE        001
# define        _FULLWIN        002
# define        _SCROLLWIN      004
# define        _FLUSH          010
# define        _FULLLINE       020
# define        _IDLINE         040
# define        _STANDOUT       0200
# define        _NOCHANGE       -1
```

_cury and *_curx* are the current (y, x) co-ordinates for the window. New characters added to the screen are added at this point. *_maxy* and *_maxx* are the maximum values allowed for (*_cury, _curx*). *_begy* and *_begx* are the starting (y, x) co-ordinates on the terminal for the window, *i.e.*, the window's home. *_cury, _curx, _maxy,* and *_maxx* are measured relative to (*_begy, _begx*), not the terminal's home.

_clear tells if a clear-screen sequence is to be generated on the next *refresh()* call. This is only meaningful for screens. The initial clear-screen for the first *refresh()* call is generated by initially setting clear to be TRUE for *curscr*, which always generates a clear-screen if set, irrelevant of the dimensions of the window involved. *_leave* is TRUE if the current (y, x) co-ordinates and the cursor are to be left after the last character changed on the terminal, or not moved if there is no change. *_scroll* is TRUE if scrolling is allowed.

---

[10] All variables not normally accessed directly by the user are named with an initial "_" to avoid conflicts with the user's variables.

_y is a pointer to an array of lines which describe the terminal. Thus:

```
_y[i]
```

is a pointer to the *i*th line, and

```
_y[i][j]
```

is the *j*th character on the *i*th line. *_flags* can have one or more values or'd into it.

For windows that are not subwindows, *_orig* is NULL . For subwindows, it points to the main window to which the window is subsidiary. *_nextp* is a pointer in a circularly linked list of all the windows which are subwindows of the same main window, plus the main window itself.

_firstch and *_lastch* are *malloc()*ed arrays which contain the index of the first and last changed characters on the line. *_ch_off* is the x offset for the window in the *_firstch* and *_lastch* arrays for this window. For main windows, this is always 0; for subwindows it is the difference between the starting point of the main window and that of the subindow, so that change markers can be set relative to the main window. This makes these markers global in scope.

All subwindows share the appropriate portions of *_y*, *_firstch*, *_lastch*, and *_insdel* with their main window.

_ENDLINE says that the end of the line for this window is also the end of a screen. **_FULLWIN** says that this window is a screen. **_SCROLLWIN** indicates that the last character of this screen is at the lower right-hand corner of the terminal; *i.e.*, if a character was put there, the terminal would scroll. **_FULLLINE** says that the width of a line is the same as the width of the terminal. If **_FLUSH** is set, it says that *fflush(stdout)* should be called at the end of each *refresh()* **_STANDOUT** says that all characters added to the screen are in standout mode. **_INSDEL** is reserved for future use, and is set by *idlok()*. *_firstch* is set to **_NOCHANGE** for lines on which there has been no change since the last *refresh()*.

1. Examples

Here we present a few examples of how to use the package. They attempt to be representative, though not comprehensive.

2. Screen Updating

The following examples are intended to demonstrate the basic structure of a program using the screen updating sections of the package. Several of the programs require calculational sections which are irrelevant of to the example, and are therefore usually not included. It is hoped that the data structure definitions give enough of an idea to allow understanding of what the relevant portions do. The rest is left as an exercise to the reader, and will not be on the final.

2.1. Twinkle

This is a moderately simple program which prints pretty patterns on the screen that might even hold your interest for 30 seconds or more. It switches between patterns of asterisks, putting them on one by one in random order, and then taking them off in the same fashion. It is more efficient to write this using only the motion optimization, as is demonstrated below.

```
/*
 * Copyright (c) 1980 Regents of the University of California.
 * All rights reserved.  The Berkeley software License Agreement
 * specifies the terms and conditions for redistribution.
 */

#ifndef lint
static char sccsid[] = "@(#)twinkle1.c          6.1 (Berkeley) 4/24/86";
#endif not lint

# include       <curses.h>
# include       <signal.h>

/*
 * the idea for this program was a product of the imagination of
 * Kurt Schoens.  Not responsible for minds lost or stolen.
 */

# define        NCOLS   80
# define        NLINES  24
# define        MAXPATTERNS     4

typedef struct {
        int     y, x;
} LOCS;

LOCS    Layout[NCOLS * NLINES];     /* current board layout */

int     Pattern,                    /* current pattern number */
        Numstars;                   /* number of stars in pattern */

char    *getenv();

int     die();
```

```
main()
{
        srand(getpid());                        /* initialize random sequence */

        initscr();
        signal(SIGINT, die);
        noecho();
        nonl();
        leaveok(stdscr, TRUE);
        scrollok(stdscr, FALSE);

        for (;;) {
                makeboard();                    /* make the board setup */
                puton('*');                     /* put on '*'s */
                puton(' ');                     /* cover up with ' 's */
        }
}

/*
 * On program exit, move the cursor to the lower left corner by
 * direct addressing, since current location is not guaranteed.
 * We lie and say we used to be at the upper right corner to guarantee
 * absolute addressing.
 */
die()
{
        signal(SIGINT, SIG_IGN);
        mvcur(0, COLS - 1, LINES - 1, 0);
        endwin();
        exit(0);
}


/*
 * Make the current board setup.  It picks a random pattern and
 * calls ison() to determine if the character is on that pattern
 * or not.
 */
makeboard()
{
        reg int         y, x;
        reg LOCS        *lp;

        Pattern = rand() % MAXPATTERNS;
        lp = Layout;
        for (y = 0; y < NLINES; y++)
                for (x = 0; x < NCOLS; x++)
                        if (ison(y, x)) {
                                lp->y = y;
                                lp->x = x;
                                lp++;
                        }
        Numstars = lp - Layout;
```

```
}

/*
 * Return TRUE if (y, x) is on the current pattern.
 */
ison(y, x)
reg int     y, x; {

        switch (Pattern) {
          case 0:           /* alternating lines */
                return !(y & 01);
          case 1:           /* box */
                if (x >= LINES && y >= NCOLS)
                        return FALSE;
                if (y < 3 || y >= NLINES - 3)
                        return TRUE;
                return (x < 3 || x >= NCOLS - 3);
          case 2:           /* holy pattern! */
                return ((x + y) & 01);
          case 3:           /* bar across center */
                return (y >= 9 && y <= 15);
        }
        /* NOTREACHED */
}

puton(ch)
reg char            ch;
{
        reg LOCS            *lp;
        reg int             r;
        reg LOCS            *end;
        LOCS                temp;

        end = &Layout[Numstars];
        for (lp = Layout; lp < end; lp++) {
                r = rand() % Numstars;
                temp = *lp;
                *lp = Layout[r];
                Layout[r] = temp;
        }

        for (lp = Layout; lp < end; lp++) {
                mvaddch(lp->y, lp->x, ch);
                refresh();
        }
}
```

2.2. Life

This program fragment models the famous computer pattern game of life (Scientific American, May, 1974). The calculational routines create a linked list of structures defining where each piece is. Nothing here claims to be optimal, merely demonstrative. This code, however, is a very good place to use the screen updating routines, as it allows them to worry

about what the last position looked like, so you don't have to. It also demonstrates some of the input routines.

```
/*
 * Copyright (c) 1980 Regents of the University of California.
 * All rights reserved.  The Berkeley software License Agreement
 * specifies the terms and conditions for redistribution.
 */

#ifndef lint
static char sccsid[] = "@(#)life.c                6.1 (Berkeley) 4/23/86";
#endif not lint

# include        <curses.h>
# include        <signal.h>

/*
 *        Run a life game.  This is a demonstration program for
 * the Screen Updating section of the -lcurses cursor package.
 */

typedef struct lst_st {                          /* linked list element */
        int                y, x;                 /* (y, x) position of piece */
        struct lst_st      *next, *last;         /* doubly linked */
} LIST;

LIST    *Head;                          /* head of linked list */

int     die();

main(ac, av)
int     ac;
char    *av[];
{
        evalargs(ac, av);                        /* evaluate arguments */

        initscr();                               /* initialize screen package */
        signal(SIGINT, die);                     /* set to restore tty stats */
        cbreak();                                /* set for char-by-char */
        noecho();                                /*
        nonl();                                  /* for optimization */

        getstart();                              /* get starting position */
        for (;;) {
                prboard();                       /* print out current board */
                update();                        /* update board position */
        }
}

/*
 * This is the routine which is called when rubout is hit.
 * It resets the tty stats to their original values.  This
 * is the normal way of leaving the program.
 */
```

```
die()
{
        signal(SIGINT, SIG_IGN);                    /* ignore rubouts */
        mvcur(0, COLS - 1, LINES - 1, 0);           /* go to bottom of screen */
        endwin();                                   /* set terminal to good state
        exit(0);
}

/*
 * Get the starting position from the user.  They keys u, i, o, j, l,
 * m, ,, and . are used for moving their relative directions from the
 * k key.  Thus, u move diagonally up to the left, , moves directly down,
 * etc.  x places a piece at the current position, " " takes it away.
 * The input can also be from a file.  The list is built after the
 * board setup is ready.
 */
getstart()
{
        reg char        c;
        reg int         x, y;
        auto char       buf[100];

        box(stdscr, '|', '_');                      /* box in the screen */
        move(1, 1);                                 /* move to upper left corner */

        for (;;) {
                refresh();                          /* print current position */
                if ((c = getch()) == 'q')
                        break;
                switch (c) {
                  case 'u':
                  case 'i':
                  case 'o':
                  case 'j':
                  case 'l':
                  case 'm':
                  case ',':
                  case '.':
                        adjustyx(c);
                        break;
                  case 'f':
                        mvaddstr(0, 0, "File name: ");
                        getstr(buf);
                        readfile(buf);
                        break;
                  case 'x':
                        addch('X');
                        break;
                  case ' ':
                        addch(' ');
                        break;
                }
        }
```

```
        if (Head != NULL)                                   /* start new list */
                dellist(Head);
        Head = malloc(sizeof (LIST));

        /*
         * loop through the screen looking for 'x's, and add a list
         * element for each one
         */
        for (y = 1; y < LINES - 1; y++)
                for (x = 1; x < COLS - 1; x++) {
                        move(y, x);
                        if (inch() == 'x')
                                addlist(y, x);
                }
}

/*
 * Print out the current board position from the linked list
 */
prboard() {

        reg LIST        *hp;

        erase();                                    /* clear out last position */
        box(stdscr, '|', '_');                      /* box in the screen */

        /*
         * go through the list adding each piece to the newly
         * blank board
         */
        for (hp = Head; hp; hp = hp->next)
                mvaddch(hp->y, hp->x, 'X');

        refresh();
}
```

3. Motion optimization

The following example shows how motion optimization is written on its own. Programs which flit from one place to another without regard for what is already there usually do not need the overhead of both space and time associated with screen updating. They should instead use motion optimization.

3.1. Twinkle

The **twinkle** program is a good candidate for simple motion optimization. Here is how it could be written (only the routines that have been changed are shown):

```
/*
 * Copyright (c) 1980 Regents of the University of California.
 * All rights reserved. The Berkeley software License Agreement
 * specifies the terms and conditions for redistribution.
 */

#ifndef lint
```

```
static char sccsid[] = "@(#)twinkle2.c          6.1 (Berkeley) 4/24/86";
#endif not lint

extern int          _putchar();

main()
{
        reg char        *sp;

        srand(getpid());                        /* initialize random sequence */

        if (isatty(0)) {
            gettmode();
            if ((sp = getenv("TERM")) != NULL)
                setterm(sp);
            signal(SIGINT, die);
        }
        else {
            printf("Need a terminal on %d\n", _tty_ch);
            exit(1);
        }
        _puts(TI);
        _puts(VS);

        noecho();
        nonl();
        tputs(CL, NLINES, _putchar);
        for (;;) {
            makeboard();                        /* make the board setup */
            puton('*');                         /* put on '*'s */
            puton(' ');                         /* cover up with ' 's */
        }
}

puton(ch)
char    ch;
{
        reg LOCS        *lp;
        reg int         r;
        reg LOCS        *end;
        LOCS            temp;
        static int      lasty, lastx;

        end = &Layout[Numstars];
        for (lp = Layout; lp < end; lp++) {
            r = rand() % Numstars;
            temp = *lp;
            *lp = Layout[r];
            Layout[r] = temp;
        }

        for (lp = Layout; lp < end; lp++)
```

```
                /* prevent scrolling */
        if (!AM || (lp->y < NLINES - 1 || lp->x < NCOLS - 1)) {
                mvcur(lasty, lastx, lp->y, lp->x);
                putchar(ch);
                lasty = lp->y;
                if ((lastx = lp->x + 1) >= NCOLS)
                        if (AM) {
                                lastx = 0;
                                lasty++;
                        }
                        else
                                lastx = NCOLS - 1;
        }
}
```

NOTES

NOTES

NOTES

NOTES

UNIX Documents

URM	User Reference Manual
	man section 1 (commands)
	man section 6 (games)
	man section 7 (miscellaneous)
USD	User Supplementary Documents
USD:1	Unix for Beginners
USD:2	Learn – Computer–Aided Instruction
USD:3	Introduction to the UNIX Shell
USD:4	Introduction to the C shell
USD:5	DC – Interactive Desk Calculator
USD:6	BC – Arbitrary Precision Desk-Calculator
USD:7	Mail Reference Manual
USD:8	MH Message Handling System
USD:9	How to Read the Network News
USD:10	How to Use USENET Effectively
USD:11	Notesfile Reference Manual
USD:12	Tutorial Introduction to "ed"
USD:13	Advanced Editing on Unix
USD:14	Edit: A Tutorial
USD:15	Introduction to Display Editing with Vi
USD:16	Ex Reference Manual (Version 3.7)
USD:17	Jove Manual for UNIX Users
USD:18	SED – A Non-interactive Text Editor
USD:19	AWK – Pattern Scanning/Processing Language
USD:20	Using the –ms Macros with Troff and Nroff
USD:21	Revised Version of –ms
USD:22	Writing Papers with *nroff* using –me
USD:23	–me Reference Manual
USD:24	NROFF/TROFF User's Manual
USD:25	TROFF Tutorial
USD:26	Typesetting Mathematics (eqn)
USD:27	Typesetting Mathematics – User's Guide
USD:28	Tbl – A Program to Format Tables
USD:29	Refer – A Bibliography System
USD:30	Some Applications of Inverted Indexes ...
USD:31	BIB – Bibliography Formatting Program
USD:32	Writing Tools – STYLE and DICTION
USD:33	A Guide to the Dungeons of Doom
USD:34	Star Trek
PRM	Programmer Reference Manual
	man sections 2 (system calls)
	man sections 3 (library routines)
	man sections 4 (devices, special files)
	man sections 5 (file formats)
PS1	Programmer Supplementary Docs, part 1
PS1:1	C Language – Reference Manual
PS1:2	Fortran 77
PS1:3	f77 I/O Library
PS1:4	Berkeley Pascal User's Manual
PS1:5	Vax Assembler Reference Manual
PS1:6	Berkeley Software Architecture Manual (4.3 Edition)
PS1:7	Introductory 4.3BSD Interprocess Communication
PS1:8	Advanced 4.3BSD Interprocess Communication
PS1:9	Lint, A C Program Checker
PS1:10	ADB Tutorial
PS1:11	Debugging with dbx
PS1:12	Make
PS1:13	Revision Control System (RCS)
PS1:14	Source Code Control System (SCCS)
PS1:15	YACC: Yet Another Compiler-Compiler
PS1:16	LEX – A Lexical Analyzer Generator
PS1:17	M4 Macro Processor
PS1:18	curses library
PS2	Programmer Supplementary Documents, part 2
PS2:1	The Unix Time–Sharing System
PS2:2	UNIX 32/V – Summary
PS2:3	Unix Programming – Second Edition
PS2:4	Unix Implementation
PS2:5	The Unix I/O System
PS2:6	Programming Language EFL
PS2:7	Berkeley FP User's Manual
PS2:8	Ratfor – Preprocessor for Rational FORTRAN
PS2:9	The FRANZ LISP Manual
PS2:10	Ingres (Version 8) Reference Manual
SMM	System Manager's Manual
	man section 8 (system administration)
SMM:1	Installing and Operating 4.3BSD
SMM:2	Building 4.3BSD Systems with *Config*
SMM:3	Using ADB to Debug the Kernel
SMM:4	Disc Quotas
SMM:5	Fsck – File System Check Program
SMM:6	Line Printer Spooler Manual
SMM:7	Sendmail Installation and Operation
SMM:8	Timed Installation and Operation
SMM:9	UUCP Implementation Description
SMM:10	USENET Version B Installation
SMM:11	Name Server Operations Guide
SMM:12	Bug Fixes and Changes in 4.3BSD
SMM:13	Changes to the Kernel in 4.3BSD
SMM:14	A Fast File System for UNIX
SMM:15	4.3BSD Networking Implementation Notes
SMM:16	Sendmail – An Internetwork Mail Router
SMM:17	On the Security of UNIX
SMM:18	Password Security – A Case History
SMM:19	A Tour Through the Portable C Compiler
SMM:20	Writing NROFF Terminal Descriptions
SMM:21	A Dial–Up Network of UNIX Systems
SMM:22	Berkeley Time Synchronization Protocol