UNIPLUS+ SYSTEM V
Programming Guide

PN: 1174-01

PREFACE

'his guide describes the C programming language supported by the JniPlus+* System V operating system. The user should have at least wo years of specialized training in computer-related fields. The user is lso expected to use UniPlus+ for system development.

'his guide contains eight chapters:

- C INTERFACE NOTES
- C LANGUAGE
- C LIBRARY
- MATH LIBRARY
- EFL: PROGRAMMING LANGUAGE
- LINT: C PROGRAM CHECKER
- UNIX IMPLEMENTATION
- UNIX I/O

:hapter 1, C INTERFACE NOTES, describes the way in which the JniSoft 68000 C programming language represents data in storage and ıow that data is passed between functions.

:hapter 2, C LANGUAGE, provides a summary of the grammar and ules of the C programming language which was used to write most of he UNIX† operating system.

:hapter 3, C LIBRARY, describes the functions and declarations that upport the C Language and how to use these functions.

UniPlus+ is a trademark of UniSoft Systems.
UNIX is a trademark of AT&T Bell Laboratories.

Chapter 4, MATH LIBRARY, describes the Math library that is supported on UniPlus$^+$.

Chapter 5, EFL: PROGRAMMING LANGUAGE, describes the programming language EFL. The reader should have a fair degree of familiarity with some procedural language.

Chapter 6, LINT: C PROGRAM CHECKER, describes a program that attempts to detect compile-time bugs and non-portable features in C programs.

Chapter 7, UNIX IMPLEMENTATION, describes the implementation of the resident UNIX kernel which includes how the system views processes, users, and programs.

Chapter 8, UNIX I/O SYSTEM, gives an overview of the I/O system and guides writers of device driver routines. The reader should have a good knowledge of the overall structure of the file system.

Throughout this document, any reference of the form **name**(1M), **name**(7), or **name**(8) refers to entries in the *UniPlus$^+$ System V Administrator's Manual*. Any reference of the form **name**(*N*) where *N* is a number 1 through 6, possibly followed by a letter, refers to entry **name** in section *N* of the *UniPlus$^+$ System V User's Manual*.

CONTENTS

Chapter 1	C INTERFACE NOTES
Chapter 2	C LANGUAGE
Chapter 3	C LIBRARY
Chapter 4	MATH LIBRARY
Chapter 5	EFL: PROGRAMMING LANGUAGE
Chapter 6	LINT: C PROGRAM CHECKER
Chapter 7	UNIX IMPLEMENTATION
Chapter 8	UNIX I/O SYSTEM

Chapter 1: C INTERFACE NOTES

CONTENTS

INTRODUCTION 1
DATA REPRESENTATIONS 2
PARAMETER PASSING IN C 3
SETTING UP THE STACK 5
ALLOCATION OF LOCAL VARIABLES AND REGISTERS 6
RETURNING FROM A FUNCTION OR SUBROUTINE . . . 7
SYSTEM CALLS 8
OPTIMIZATIONS 8
USE OF REGISTER VARIABLES 9

Chapter 1

C INTERFACE NOTES

INTRODUCTION

This chapter describes the way in which the UniSoft 68000 C programming language represents data in storage, and how that data is passed between functions. Also described is the environment of a function, and the calling mechanism for functions.

The information in this chapter is intended for programmers who have detailed knowledge of the interface mechanisms in order to match C code with the assembler. It is also intended for those who wish to write new system functions or mathematical functions.

When a C program is compiled and assembled, the program is split into three parts. These are:

.text The executable code of the program.

.data The initialized data area. This contains literal constants, character strings, and so on.

.bss The uninitialized data areas.

These three parts of the program appear in the above order. The compiler/assembler combination produces the first two. The loader actually generates the **.bss** area at load time.

The **.bss** area is cleared to zero (0) by the loader at load time. This is a feature of the system and can be relied upon.

During execution of a program, the stack area contains indeterminate data. In other words, its previous contents (if any) cannot be relied upon.

DATA REPRESENTATIONS

In general, all data elements of whatever size are stored such that their least significant bit is in the highest addressed byte and their most significant bit is in the lowest addressed byte. The list below describes the representation of data.

char 6 Values of type *char* occupy 8 bits. Such values can be aligned on any byte boundary.

short 6 Values of type *short* occupy 16 bits. Values of type *short* are aligned on word (16-bit) address boundaries.

long 6 Values of type *long* occupy 32 bits. A *long* value is the same as an *int* value in 68000 C. Values of this type are aligned on word (16-bit) boundaries.

float 6 Values of type *float* occupy 32 bits. All *float* values are automatically converted to type *double* for computation purposes. Values of this type are aligned on word (16-bit) boundaries. A *float* value consists of a sign bit, followed by an 8-bit biased exponent, followed by a 23-bit mantissa.

double 6 Values of type *double* occupy 64 bits. Values of this type are aligned on word (16-bit) boundaries. A *double* value consists of a sign bit, followed by an 8-bit biased exponent, followed by a 55-bit mantissa.

pointers 6 All *pointers* are represented as long (32-bit) values. Pointers are aligned on word (16-bit) boundaries.

arrays 6 The base address of an *array* value is always aligned on a word (16-bit) address boundary.

Elements of an array are stored contiguously, one after the other. Elements of multi-dimensional arrays are stored in row-major order. That is, the last dimension of an array varies the fastest.

When a multi-dimensional array is declared, it is possible to omit the size specification for the last dimension. In such a case, what is allocated is actually an array of pointers to the elements of the last dimension.

structures and unions

Within structures and unions, it is possible to obtain unfilled holes of size *char*. This is due to the compiler rounding addresses up to 16-bit boundaries to accommodate word-aligned elements.

This situation can best be demonstrated by an example. Consider the following structure:

```
struct {
    int   x;        /* This is a 32-bit element      */
    char  y;        /* Takes up a single byte        */
    short z;        /* Aligned to a 16-bit boundary  */
};
```

The total number of bytes declared above is seven: four for the *int*, one for the *char*, and two for the *short*.

In reality, the "z" field which is a *short* will be aligned on a 16-bit boundary by the C compiler. In this case, the compiler inserts a hole after the *char* element "y", to align the *short* element "z". The net effect of these machinations is a structure that behaves like this:

```
struct {
    int   x;        /* This is a 32-bit element      */
    char  y;        /* Takes up a single byte        */
    char  dummy;    /* Fills the structure           */
    short z;        /* Aligned to a 16-bit boundary  */
};
```

The C compiler **never** reorders any parts of a structure.

Similar considerations apply to arrays of structures or unions. Each element of an array (other than an array of *char*) begins on a 16-bit boundary.

For a detailed treatment of data storage, consult *The C Programming Language* by Kernighan and Ritchie.

PARAMETER PASSING IN C

The C programming language is unique in that it really has only functions. The effect of a subroutine is achieved simply by having a function which does not return a value.

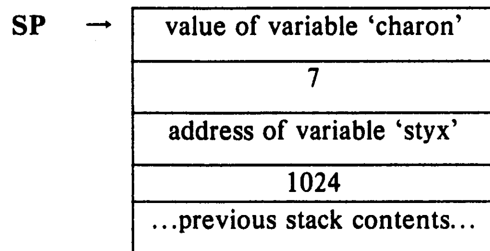
Another unique feature of C is that parameters to functions are always passed by value. The C programming language has no concept of declaring parameters to be passed by reference, as there is in languages such as Pascal. In order to pass a parameter by reference in a C program, the programmer must explicitly pass the address of the parameter. The called function must be aware that it is receiving an address instead of a value, and the appropriate code must be present to handle that case.

When a function is called, its parameters (if any) are evaluated and are then pushed onto the stack in reverse order. All parameters are pushed onto the stack as 32-bit longs. If a parameter is shorter than 32 bits, it is expanded to a 32-bit value with sign-extension, if necessary. The calling procedure is responsible for popping the parameters off the stack.

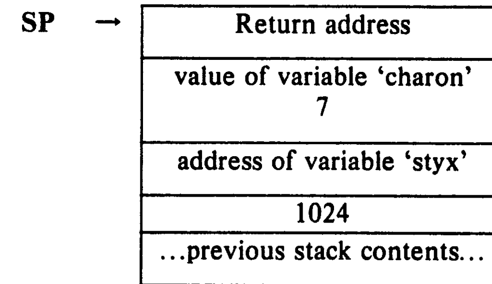
Consider a C function call like this:

```
ferry (charon, 7, &styx, 1<<10);
```

After evaluation, but just before the call, the stack looks like this:



Functions are called by issuing either a "bsr" instruction or a "jsr" instruction, depending upon whether the callee is within a 16-bit addressing range or not, and whether the C optimizer was used. The "bsr" or "jsr" instruction pushes the return address upon the stack, and then branches to the indicated function. After the call, on entry to the function, the stack looks like this:



In each function, register A6 is used as a stack frame base. The stack location referenced by A6 contains the return address.

SETTING UP THE STACK

Upon entry into the function, the prolog code is executed. The prolog code allocates enough space on the stack for the local variables, plus enough space to save any registers that this function uses. The prolog code then ensures that there is enough stack space available for executing the function. If there is not enough space, the system grows the stack to allot more space. The prolog code looks like this:

```
link      a6,#-.F1
tstb      sp@(-page_size)
moveml    #.S1,a6@(-.F1)
```

The ".F1" constant is the size of the stack frame for the local variables, plus four bytes for each register variable.

The "page_size" constant is an implementation dependent constant. It is used to probe the stack region at some place below the current stack top. If the probe generates a trap, the system grows the stack by an amount sufficient to include the probe address.

Finally, the ".S1" constant is a mask to determine which registers need to be saved on the stack for this particular function. This is, of course, dependent on the register variables that the programmer declared for that particular routine.

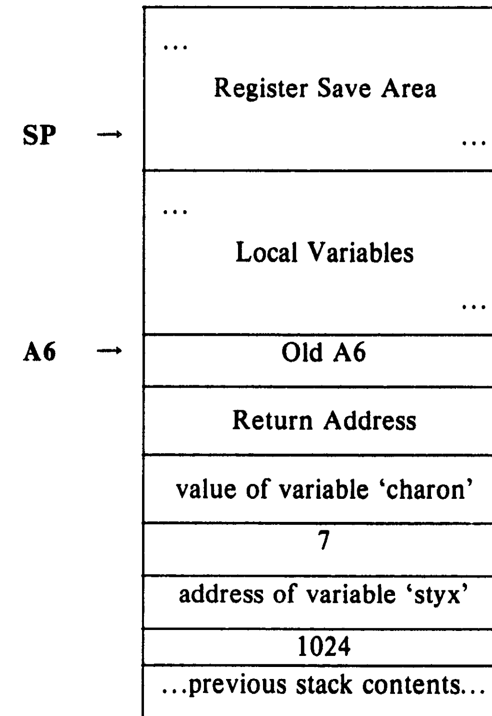
ALLOCATION OF LOCAL VARIABLES AND REGISTERS

A total of nine registers are available for register variables. Five of these are the 68000 data (D) registers, and four are the 68000 address (A) registers. The available A registers are A2 through A5. The available D registers are D3 through D7.

Any variable declared as a pointer variable is always allocated to an address register. Non-pointer variables are assigned to data registers. Register variables are allocated to registers in the order in which they are declared in the C source program, starting at the high end (A5 or D7) of the appropriate type of register.

If there are more register variables of either kind than there are registers to accommodate them, the remaining variables are allocated on the stack as local variables, just as if the register attribute had never been given in the declaration.

Upon completion of the prolog code, the stack then looks like this:



RETURNING FROM A FUNCTION OR SUBROUTINE

Upon reaching a "return" statement, either explicit or implicit, the function executes the epilog code. If the function has a return value, generated from a

```
return(expression);
```

statement, the value of the expression (which is synonymous with the value of the function) is placed in register D0. The epilog code is then executed to effect a return from the function:

```
moveml    a6@(-.F1),#.S1
unlk      a6
rts
```

The "moveml" instruction restores any registers which were saved during the prolog. The stack frame base pointer in A6 is then put back to the point where A6 once again points to the return address. The

function is then exited via the "rts" instruction, which pops the stack to the state it was in prior to the original call, and then returns to the function that called it.

SYSTEM CALLS

The C compiler generates code for system calls in the following way:

- The system call number is placed in register D0.
- The first parameter is placed in register A0; the second parameter goes in register D1; the third parameter is placed in register A1; and the fourth parameter is placed in register D2.
- A "TRAP #0" instruction is executed.

The C compiler sometimes generates code which uses register D2, so if your code uses D2, you must save it before executing the system call. On return from the system call, errors are signaled by the carry flag being set. The C interface to the system calls typically returns a −1 on error as the carry flag cannot be tested from C.

OPTIMIZATIONS

This section describes some of the ways in which the programmer can optimize the use of the C language.

The C compiler can be run to optimize the code it generates, making that code both compact and fast. Using a C command line as follows:

cc −O *file*

generates optimized code. The option for optimized code generation is an upper-case "O".

If a C program contains a "do" loop of the form:

```
register short x;
x = 10;
do {
        statement
} while (--x != -1);
```


Such a loop is optimized to use the "dbra" instruction, resulting in faster execution.

USE OF REGISTER VARIABLES

The decision as to whether to declare a variable in a register depends on the number of times that variable is referenced in the function. If a variable is used more than twice in a function, it can be declared as a register variable. If a variable is used less than twice in a function, it is not useful to declare it as a register variable because the amount of time spent saving and restoring that register is more than the time saved in using a register instead of a location on the stack.

Chapter 2: C LANGUAGE

CONTENTS

LEXICAL CONVENTIONS 1
- Comments 1
- Identifiers (Names) 1
- Keywords 2
- Constants 2
  - Integer Constants 2
  - Explicit Long Constants 2
  - Character Constants 3
  - Floating Constants 3
  - Enumeration Constants 4
- Strings 4
- Hardware Characteristics 4

SYNTAX NOTATION 6

NAMES 6
- Storage Class 7
- Type 7

OBJECTS AND LVALUES 9

CONVERSIONS 9
- Characters and Integers 9
- Float and Double 10
- Floating and Integral 10
- Pointers and Integers 10
- Unsigned 10
- Arithmetic Conversions 11

EXPRESSIONS 12
- Primary Expressions 12
- Unary Operators 15
- Multiplicative Operators 17
- Additive Operators 18
- Shift Operators 19
- Relational Operators 19
- Equality Operators 20
- Bitwise AND Operator 20
- Bitwise Exclusive OR Operator 20

Bitwise Inclusive OR Operator 21
Logical AND Operator 21
Logical OR Operator 21
Conditional Operator 22
Assignment Operators 22
Comma Operator 23

DECLARATIONS 24
Storage Class Specifiers 24
Type Specifiers 25
Declarators 26
Meaning of Declarators 27
Structure and Union Declarations 29
Enumeration Declarations 33
Initialization 34
Type Names 37
Typedef 38

STATEMENTS 39
Expression Statement 39
Compound Statement or Block 39
Conditional Statement 40
While Statement 40
Do Statement 40
For Statement 41
Switch Statement 41
Break Statement 42
Continue Statement 43
Return Statement 43
Goto Statement 44
Labeled Statement 44
Null Statement 44

EXTERNAL DEFINITIONS 44
External Function Definitions 45
External Data Definitions 46

SCOPE RULES 47
Lexical Scope 47
Scope of Externals 48

COMPILER CONTROL LINES 49
Token Replacement 49
File Inclusion 50
Conditional Compilation 51

Line Control 52

IMPLICIT DECLARATIONS 53

TYPES REVISITED 53
Structures and Unions 53
Functions 54
Arrays, Pointers, and Subscripting 55
Explicit Pointer Conversions 56

CONSTANT EXPRESSIONS 57

PORTABILITY CONSIDERATIONS 58

SYNTAX SUMMARY 59
Expressions 60
Declarations 62
Statements 65
External definitions 65
Preprocessor 66

LIST OF FIGURES

Figure 2.1. 68000 Hardware Characteristics 5

Chapter 2

C LANGUAGE

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.

Comments

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

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:

68000	7 characters, 2 cases

Keywords

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

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

Some implementations also reserve the words **fortran** and **asm**.

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."

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**.

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.

C LANGUAGE

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**.

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**.

Enumeration Constants

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

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.

Hardware Characteristics

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

	68000 (ASCII)
char	8 bits
int	32
short	16
long	32
float	32
double	64
float range	$\pm 10^{\pm 38}$
double range	$\pm 10^{\pm 307}$

Figure 2.1. 68000 Hardware Characteristics

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".

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.

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.

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 allot storage.

Unsigned integers, declared **unsigned,** obey the laws of arithmetic modulo 2^n where *n* is the number of bits in the representation.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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."

1. 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**.

2. Then, if either operand is **double,** the other is converted to **double** and that is the type of the result.

3. Otherwise, if either operand is **unsigned long**, the other is converted to **unsigned long** and that is the type of the result.

4. Otherwise, if either operand is **long,** the other is converted to **long** and that is the type of the result.

5. 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.

6. Otherwise, if either operand is **unsigned,** the other is converted to **unsigned** and that is the type of the result.

7. Otherwise, both operands must be **int**, and that is the type of the result.

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.

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."

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**.

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).

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.

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.

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.

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.

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.

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.

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.

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.

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**.

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**.

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.

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.

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.

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.

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. 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.

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

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."

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.

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**.

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.

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.

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**.

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}.

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.

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."

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.

STATEMENTS

Except as indicated, statements are executed in sequence.

Expression Statement

Most statements are expression statements, which have the form

expression ;

Usually expression statements are assignments or function calls.

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.

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**.

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.

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.

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.

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.

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.

Continue Statement

The statement

continue ;

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

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

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

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.

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.

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."

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**.

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.

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 ...."

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**.

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.

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**.

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**.

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.

Token Replacement

A compiler-control line of the form

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

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

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

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

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

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

```
#define TABSIZE 100

int table[TABSIZE];
```

A control line of the form

#undef *identifier*

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

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

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.

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.

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.

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**."

TYPES REVISITED

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

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) ...
```

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 **(**.

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.

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 *alloc();
double *dp;

dp = (double *) alloc(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.

On the 68000, pointers are 32-bits long and measure bytes. The **char**'s have no alignment requirements; everything else must have an even address.

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.

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.

SYNTAX SUMMARY

This summary of C syntax is intended more for aiding comprehension than as an exact statement of the language.

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

C LANGUAGE

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.

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

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

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
;

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* ;

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*"

Chapter 3: C LIBRARIES

CONTENTS

GENERAL 1
  Including Functions 2
  Including Declarations 2

THE C LIBRARY 3
  Input/Output Control 3
  File Access Functions 4
  File Status Functions 5
  Input Functions 5
  Output Functions 6
  Miscellaneous Functions 7
  String Manipulation Functions 8
  Character Manipulation 9
  Character Testing Functions 10
  Character Translation Functions 11
  Time Functions 11
  Miscellaneous Functions 12
  Numerical Conversion 13
  DES Algorithm Access 13
  Group File Access 14
  Password File Access 15
  Parameter Access 15
  Hash Table Management 16
  Binary Tree Management 17
  Table Management 17
  Memory Allocation 18
  Pseudorandom Number Generation 19
  Signal Handling Functions 20
  Miscellaneous 21

Chapter 3
C LIBRARIES

GENERAL

This chapter describes the C library that is supported on the UniPlus+ operating system. A library is a collection of related functions and/or declarations that simplify programming effort by linking only what is needed, allowing use of locally produced functions, etc. All of the functions described are also described in Section 3 of the *UniPlus+ System V User's Manual*. Most of the declarations described are in Section 5 of the *UniPlus+ System V User's Manual*.

The two main libraries on the UniPlus+ system are:

C library — This is the basic library for C language programs. The C library is composed of functions and declarations used for file access, string testing and manipulation, character testing and manipulation, memory allocation, and other functions. This library is described later in this chapter.

Math library — This library provides exponential, bessel functions, logarithmic, hyperbolic, and trigonometric functions. This library is described in Chapter 4.

Some libraries consist of two portions — functions and declarations. In some cases, the user must request that the functions (and/or declarations) of a specific library be included in a program being compiled. In other cases, the functions (and/or declarations) are included automatically.

Including Functions

When a program is being compiled, the compiler will automatically search the C language library to locate and include functions that are used in the program. This is the case only for the C library and no other library. In order for the compiler to locate and include functions from other libraries, the user must specify these libraries on the command line for the compiler. For example, when using functions of the math library, the user must request that the math library be searched by including the argument **-lm** on the command line, such as:

```
cc file.c -lm
```

The argument **-lm** must come after all files that reference functions in the math library in order for the link editor to know which functions to include in the a.out file.

This method should be used for all functions that are not part of the C language library.

Including Declarations

Some functions require a set of declarations in order to operate properly. A set of declarations is stored in a file under the */usr/include* directory. These files are referred to as *header files*. In order to include a certain header file, the user must specify this request within the C language program. The request is in the form:

```
#include <file.h>
```

where *file.h* is the name of the file. Since the header files define the type of the functions and various preprocessor constants, they must be included before invoking the functions they declare.

The remainder of this chapter describes the functions and header files of the C Library. The description of the library begins with the actions required by the user to include the functions and/or header files in a program being compiled (if any). Following the description

of the actions required is information in three-column format of the form:

function **reference**(N) Brief description.

The functions are grouped by type while the reference refers to section 'N' in the *UniPlus⁺ System V User's Manual*. Following this, are descriptions of the header files associated with these functions (if any).

THE C LIBRARY

The C library consists of several types of functions. All the functions of the C library are loaded automatically by the compiler. Various declarations must sometimes be included by the user as required. The functions of the C library are divided into the following types:

- Input/output control
- String manipulation
- Character manipulation
- Time functions
- Miscellaneous functions.

Input/Output Control

These functions of the C library are automatically included as needed during the compiling of a C language program. No command line request is needed.

The header file required by the input/output functions should be included in the program being compiled. This is accomplished by including the line:

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

near the beginning of each file that references an input or output function.

The input/output functions are grouped into the following categories:

- File access
- File status
- Input
- Output
- Miscellaneous.

File Access Functions

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
fclose	**fclose**(3S)	Close an open stream.
fdopen	**fopen**(3S)	Associate stream with an **open**(2) ed file.
fileno	**ferror**(3S)	File descriptor associated with an open stream.
fopen	**fopen**(3S)	Open a file with specified permissions. **Fopen** returns a pointer to a stream which is used in subsequent references to the file.
freopen	**fopen**(3S)	Substitute named file in place of open stream.
fseek	**fseek**(3S)	Reposition the file pointer.
pclose	**popen**(3S)	Close a stream opened by **popen**.
popen	**popen**(3S)	Create pipe as a stream between calling process and command.
rewind	**fseek**(3S)	Reposition file pointer at beginning of file.
setbuf	**setbuf**(3S)	Assign buffering to stream.

File Status Functions

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
clearerr	**ferror**(3S)	Reset error condition on stream.
feof	**ferror**(3S)	Test for "end of file" on stream.
ferror	**ferror**(3S)	Test for error condition on stream.
ftell	**fseek**(3S)	Return current position in the file.

Input Functions

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
fgetc	**getc**(3S)	True function for **getc** (3S).
fgets	**gets**(3S)	Read string from stream.
fread	**fread**(3S)	General buffered read from stream.

C LIBRARIES

fscanf	**scanf**(3S)	Formatted read from stream.
getc	**getc**(3S)	Read character from stream.
getchar	**getc**(3S)	Read character from standard input.
gets	**gets**(3S)	Read string from standard input.
getw	**getc**(3S)	Read word from stream.
scanf	**scanf**(3S)	Read using format from standard input.
sscanf	**scanf**(3S)	Formatted from string.
ungetc	**ungetc**(3S)	Put back one character on stream.

Output Functions

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
fflush	**fclose**(3S)	Write all currently buffered characters from stream.
fprintf	**printf**(3S)	Formatted write to stream.
fputc	**putc**(3S)	True function for **putc** (3S).
fputs	**puts**(3S)	Write string to stream.
fwrite	**fread**(3S)	General buffered write to stream.
printf	**printf**(3S)	Print using format to standard output.
putc	**putc**(3S)	Write character to standard output.
putchar	**putc**(3S)	Write character to standard output.
puts	**puts**(3S)	Write string to standard output.
putw	**putc**(3S)	Write word to stream.
sprintf	**printf**(3S)	Formatted write to string.

Miscellaneous Functions

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
ctermid	**ctermid**(3S)	Return file name for controlling terminal.
cuserid	**cuserid**(3S)	Return login name for owner of current process.
system	**system**(3S)	Execute shell command.

tempnam	**tmpnam**(3S)	Create temporary file name using directory and prefix.
tmpnam	**tmpnam**(3S)	Create temporary file name.
tmpfile	**tmpfile**(3S)	Create temporary file.

String Manipulation Functions

These functions are used to locate characters within a string, copy, concatenate, and compare strings. These functions are automatically located and loaded during the compiling of a C language program. No command line request is needed since these functions are part of the C library. The string manipulation functions are declared in a header file that may be included in the program being compiled. This is accomplished by including the line:

```
#include <string.h>
```

near the beginning of each file that uses one of these functions.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
strcat	**string**(3C)	Concatenate two strings.
strchr	**string**(3C)	Search string for character.
strcmp	**string**(3C)	Compares two strings.
strcpy	**string**(3C)	Copy string.
strcspn	**string**(3C)	Length of initial string not containing set of characters.
strlen	**string**(3C)	Length of string.
strncat	**string**(3C)	Concatenate two strings with a maximum length.
strncmp	**string**(3C)	Compares two strings with a maximum length.
strncpy	**string**(3C)	Copy string over string with a maximum length.
strpbrk	**string**(3C)	Search string for any set of characters.
strrchr	**string**(3C)	Search string backwards for character.
strspn	**string**(3C)	Length of initial string containing set of characters.
strtok	**string**(3C)	Search string for token separated by any of a set of characters.

Character Manipulation

The following functions and declarations are used for testing and translating ASCII characters. These functions are located and loaded automatically during the compiling of a C language program. No command line request is needed since these functions are part of the C library.

The declarations associated with these functions should be included in the program being compiled. This is accomplished by including the line:

```
#include <ctype.h>
```

near the beginning of the file being compiled.

Character Testing Functions

These functions can be used to identify characters as uppercase or lowercase letters, digits, punctuation, etc.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
isalnum	**ctype**(3C)	Is character alphanumeric?
isalpha	**ctype**(3C)	Is character alphabetic?
isascii	**ctype**(3C)	Is integer ASCII character?
iscntrl	**ctype**(3C)	Is character a control character?
isdigit	**ctype**(3C)	Is character a digit?
isgraph	**ctype**(3C)	Is character a printable character?
islower	**ctype**(3C)	Is character a lowercase letter?
isprint	**ctype**(3C)	Is character a printing character including space?
ispunct	**ctype**(3C)	Is character a punctuation character?
isspace	**ctype**(3C)	Is character a white space character?
isupper	**ctype**(3C)	Is character an uppercase letter?
isxdigit	**ctype**(3C)	Is character a hex digit?

Character Translation Functions

These functions provide translation of uppercase to lowercase, lowercase to uppercase, and integer to ASCII.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
toascii	**conv**(3C)	Convert integer to ASCII character.
tolower	**conv**(3C)	Convert character to lowercase.
toupper	**conv**(3C)	Convert character to uppercase.

Time Functions

These functions are used for accessing and reformatting the systems idea of the current date and time. These functions are located and loaded automatically during the compiling of a C language program. No command line request is needed since these functions are part of the C library.

The header file associated with these functions should be included in the program being compiled. This is accomplished by including the line:

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

near the beginning of any file using the time functions.

These functions (except **tzset**) convert a time such as returned by **time**(2).

FUNCTION	REFERENCE	BRIEF DESCRIPTION
asctime	**ctime**(3C)	Return string representation of date and time.
ctime	**ctime**(3C)	Return string representation of date and time, given integer form.
gmtime	**ctime**(3C)	Return Greenwich Mean Time.
localtime	**ctime**(3C)	Return local time.
tzset	**ctime**(3C)	Set time zone field from environment variable.

Miscellaneous Functions

These functions support a wide variety of operations. Some of these are numerical conversion, password file and group file access, memory allocation, random number generation, and table management. These functions are automatically located and included in a program being compiled. No command line request is needed since these functions are part of the C library.

Some of these functions require declarations to be included. These are described following the descriptions of the functions.

Numerical Conversion

The following functions perform numerical conversion.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
a64l	**a64l**(3C)	Convert string to base 64 ASCII.
atof	**atof**(3C)	Convert string to floating.
atoi	**atof**(3C)	Convert string to integer.
atol	**atof**(3C)	Convert string to long.
frexp	**frexp**(3C)	Split floating into mantissa and exponent.
l3tol	**l3tol**(3C)	Convert 3-byte integer to long.
ltol3	**l3tol**(3C)	Convert long to 3-byte integer.
ldexp	**frexp**(3C)	Combine mantissa and exponent.
l64a	**a64l**(3C)	Convert base 64 ASCII to string.
modf	**frexp**(3C)	Split mantissa into integer and fraction.

DES Algorithm Access

The following functions allow access to the Data Encryption Standard (DES) algorithm used on the UNIX operating system. The DES algorithm is implemented with variations to frustrate use of hardware implementations of the DES for key search.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
crypt	**crypt**(3C)	Encode string.
encrypt	**crypt**(3C)	Encode/decode string of 0s and 1s.
setkey	**crypt**(3C)	Initialize for subsequent use of **encrypt**.

Group File Access

The following functions are used to obtain entries from the group file. Declarations for these functions must be included in the program being compiled with the line:

```
#include <grp.h>
```

FUNCTION	REFERENCE	BRIEF DESCRIPTION
endgrent	**getgrent**(3C)	Close group file being processed.
getgrent	**getgrent**(3C)	Get next group file entry.
getgrgid	**getgrent**(3C)	Return next group with matching gid.
getgrnam	**getgrent**(3C)	Return next group with matching name.
setgrent	**getgrent**(3C)	Rewind group file being processed.
fgetgrent	**getgrent**(3C)	Get next group file entry from a specified file.

Password File Access

These functions are used to search and access information stored in the password file (/etc/passwd). Some functions require declarations that can be included in the program being compiled by adding the line:

```
#include <pwd.h>
```

FUNCTION	REFERENCE	BRIEF DESCRIPTION
endpwent	**getpwent**(3C)	Close password file being processed.
getpw	**getpw**(3C)	Search password file for uid.
getpwent	**getpwent**(3C)	Get next password file entry.
getpwnam	**getpwent**(3C)	Return next entry with matching name.
getpwuid	**getpwent**(3C)	Return next entry with matching uid.
putpwent	**putpwent**(3C)	Write entry on stream.
setpwent	**getpwent**(3C)	Rewind password file being accessed.
fgetpwent	**getpwent**(3C)	Get next password file entry from a specified file.

Parameter Access

The following functions provide access to several different types of parameters. None require any declarations.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
getopt	**getopt**(3C)	Get next option from option list.
getcwd	**getcwd**(3C)	Return string representation of current working directory.
getenv	**getenv**(3C)	Return string value associated with environment variable.
getpass	**getpass**(3C)	Read string from terminal without echoing.

Hash Table Management

The following functions are used to manage hash search tables. The header file associated with these functions should be included in the program being compiled. This is accomplished by including the line:

```
#include <search.h>
```

near the beginning of any file using the search functions.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
hcreate	**hsearch**(3C)	Create hash table.
hdestroy	**hsearch**(3C)	Destroy hash table.
hsearch	**hsearch**(3C)	Search hash table for entry.

Binary Tree Management

The following functions are used to manage a binary tree. The header file associated with these functions should be included in the program being compiled. This is accomplished by including the line:

```
#include <search.h>
```

near the beginning of any file using the search functions.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
tdelete	**tsearch**(3C)	Deletes nodes from binary tree.
tsearch	**tsearch**(3C)	Look for and add element to binary tree.
twalk	**tsearch**(3C)	Walk binary tree.

Table Management

The following functions are used to manage a table. Since none of these functions allocate storage, sufficient memory must be allocated before using these functions. The header file associated with these functions should be included in the program being compiled. This is accomplished by including the line:

```
#include <search.h>
```

near the beginning of any file using the search functions.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
bsearch	**bsearch**(3C)	Search table using binary search.
lsearch	**lsearch**(3C)	Look for and add element in binary tree.
qsort	**qsort**(3C)	Sort table using quick-sort algorithm.

Memory Allocation

The following functions provide a means by which memory can be dynamically allocated or freed.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
calloc	**malloc**(3C)	Allocate zeroed storage.
free	**malloc**(3C)	Free previously allocated storage.
malloc	**malloc**(3C)	Allocate storage.
realloc	**malloc**(3C)	Change size of allocated storage.

Pseudorandom Number Generation

The following functions are used to generate pseudorandom numbers. The functions that end with **48** are a family of interfaces to a pseudorandom number generator based upon the linear congruent algorithm and 48-bit integer arithmetic. The **rand** and **srand** functions provide an interface to a multiplicative congruential random number generator with period of 232.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
drand48	**drand48**(3C)	Random double over the interval [0 to 1).
lcong48	**drand48**(3C)	Set parameters for **drand48**, **lrand48**, and **mrand48**.
lrand48	**drand48**(3C)	Random long over the interval [0 to 2^{31}).
mrand48	**drand48**(3C)	Random long over the interval [-2^{31} to 2^{31}).
rand	**rand**(3C)	Random integer over the interval [0 to 32767).

seed48	**drand48**(3C)	Seed the generator for **drand48**, **lrand48**, and **mrand48**.
srand	**rand**(3C)	Seed the generator for **rand**.
srand48	**drand48**(3C)	Seed the generator for **drand48**, **lrand48**, and **mrand48** using a long.

Signal Handling Functions

The functions **gsignal** and **ssignal** implement a software facility similar to **signal**(2) in the + *System V User's Manual*. This facility enables users to indicate the disposition of error conditions and allows users to handle signals for their own purposes. The declarations associated with these functions can be included in the program being complied by the line

```
#include <signal.h>
```

These declarations define ASCII names for the 15 software signals.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
gsignal	**ssignal**(3C)	Send a software signal.
ssignal	**ssignal**(3C)	Arrange for handling of software signals.
ttyname	**ttyname**(3C)	Return pathname of terminal associated with integer file descriptor.

Miscellaneous

The following functions do not fall into any previously described category.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
abort	**abort**(3C)	Cause an IOT signal to be sent to the process.
abs	**abs**(3C)	Return the absolute integer value.
ecvt	**ecvt**(3C)	Convert double to string.
fcvt	**ecvt**(3C)	Convert double to string using Fortran Format.
gcvt	**ecvt**(3C)	Convert double to string using Fortran F or E format.
isatty	**ttyname**(3C)	Test whether integer file descriptor is associated with a terminal.
mktemp	**mktemp**(3C)	Create file name using template.
monitor	**monitor**(3C)	Cause process to record a histogram of program counter location.
swab	**swab**(3C)	Swap and copy bytes.

Chapter 4: MATH LIBRARY

CONTENTS

GENERAL 1

THE MATH LIBRARY 1
  Trigonometric Functions 2
  Bessel Functions 3
  Hyperbolic Functions 3
  Miscellaneous Functions 3

Chapter 4

MATH LIBRARY

GENERAL

This chapter describes the Math Library that is supported on the UNIX operating system. A library is a collection of related functions and/or declarations that simplify programming effort. All of the functions described are also described in Part 3 of the *UniPlus+ System V User's Manual*; most of the declarations described are in Part 5. The two main libraries on the UNIX system are:

C library — This is the basic library for C language programs. The C library is composed of functions and declarations used for file access, string testing and manipulation, character testing and manipulation, memory allocation, and other functions. This library is described in Chapter 3.

Math library — This library provides exponential, bessel functions, logarithmic, hyperbolic, and trigonometric functions. This library is described in this chapter.

THE MATH LIBRARY

The math library consists of functions and a header file. The functions are located and loaded during the compiling of a C language program by a command line request. The form of this request is:

cc *file* **−lm**

which causes the link editor to search the math library. In addition to the request to load the functions, the header file of the math

library should be included in the program being compiled. This is accomplished by including the line:

```
#include <math.h>
```

near the beginning of the (first) file being compiled.

The functions are grouped into the following categories:

- Trigonometric functions
- Bessel functions
- Hyperbolic functions
- Miscellaneous functions.

Trigonometric Functions

These functions are used to compute angles (in radian measure), sines, cosines, and tangents. All of these values are expressed in double precision.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
acos	**trig**(3M)	Return arc cosine.
asin	**trig**(3M)	Return arc sine.
atan	**trig**(3M)	Return arc tangent.
atan2	**trig**(3M)	Return arc tangent of a ratio.
cos	**trig**(3M)	Return cosine.
sin	**trig**(3M)	Return sine.
tan	**trig**(3M)	Return tangent.

Bessel Functions

These functions calculate bessel functions of the first and second kinds of several orders for real values. The bessel functions are **j0**, **j1**, **jn**, **y0**, **y1**, and **yn**. The functions are located in section **bessel**(3M).

Hyperbolic Functions

These functions are used to compute the hyperbolic sine, cosine, and tangent for real values.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
cosh	**sinh**(3M)	Return hyperbolic cosine.
sinh	**sinh**(3M)	Return hyperbolic sine.
tanh	**sinh**(3M)	Return hyperbolic tangent.

Miscellaneous Functions

These functions cover a wide variety of operations, such as natural logarithm, exponential, and absolute value. In addition, several are provided to truncate the integer portion of double precision numbers.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
ceil	**floor**(3M)	Returns the smallest integer not less than a given value.
exp	**exp**(3M)	Returns the exponential function of a given value.

fabs	**floor**(3M)	Returns the absolute value of a given value.
floor	**floor**(3M)	Returns the largest integer not greater than a given value.
fmod	**floor**(3M)	Returns the remainder produced by the division of two given values.
gamma	**gamma**(3M)	Returns the natural log of the absolute value of the result of applying the gamma function to a given value.
hypot	**hypot**(3M)	Return the square root of the sum of the squares of two numbers.
log	**exp**(3M)	Returns the natural logarithm of a given value.
log10	**exp**(3M)	Returns the logarithm base ten of a given value.
matherr	**matherr**(3M)	Error-handling function.
pow	**exp**(3M)	Returns the result of a given value raised to another given value.
sqrt	**exp**(3M)	Returns the square root of a given value.

Chapter 5: EFL: PROGRAMMING LANGUAGE

CONTENTS

INTRODUCTION 1

LEXICAL FORM 2
- Character Set 2
- Lines 2
  - White Space 3
  - Comments 3
  - Include Files 3
  - Continuation 3
  - Multiple Statements on a Line 4
- Tokens 4
  - Identifiers 4
  - Strings 5
  - Integer Constants 5
  - Floating Point Constants 6
  - Punctuation 6
  - Operators 6
- Macros 7

PROGRAM FORM 8
- Files 8
- Procedures 8
- Blocks 8
- Statements 9
- Labels 10

DATA TYPES AND VARIABLES 10
- Basic Types 10
- Constants 11
- Variables 12
  - Storage Class 12
  - Scope of Names 12
  - Precision 12
- Arrays 12
- Structures 13

EXPRESSIONS 13
- Primaries 14
  - Constants 14

Variables . . . 14
Array Elements . . . 14
Structure Members . . . 15
Procedure Invocations . . . 15
Input/Output Expressions . . . 16
Coercions . . . 16
Sizes . . . 17
Parentheses . . . 17
Unary Operators . . . 18
Arithmetic . . . 18
Logical . . . 18
Binary Operators . . . 18
Arithmetic . . . 18
Logical . . . 19
Relational Operators . . . 20
Assignment Operators . . . 21
Dynamic Structures . . . 21
Repetition Operator . . . 22
Constant Expressions . . . 22

DECLARATIONS . . . 22
Syntax . . . 22
Attributes . . . 23
Basic Types . . . 23
Arrays . . . 23
Structures . . . 24
Precision . . . 25
Common . . . 25
External . . . 25
Variable List . . . 26
The Initial Statement . . . 26

EXECUTABLE STATEMENTS . . . 26
Expression Statements . . . 27
Subroutine Call . . . 27
Assignment Statements . . . 27
Blocks . . . 27
Test Statements . . . 28
If Statement . . . 28
If-Else . . . 28
Select Statement . . . 29
Loops . . . 30
While Statement . . . 30

For Statement . . . 30
Repeat Statement . . . 31
Repeat ... Until Statement . . . 31
Do Loop . . . 32
Branch Statements . . . 33
Goto Statement . . . 33
Break Statement . . . 34
Next Statement . . . 35
Return . . . 35
Input/Output Statements . . . 35
Input/Output Units . . . 36
Binary Input/Output . . . 36
Formatted Input/Output . . . 36
Iolists . . . 37
Formats . . . 37
Manipulation Statements . . . 38

PROCEDURES . . . 38
Procedures Statement . . . 39
End Statement . . . 39
Argument Association . . . 39
Execution and Return Values . . . 40
Known Functions . . . 40
Minimum and Maximum Functions . . . 40
Absolute Value . . . 40
Elementary Functions . . . 41
Other Generic Functions . . . 41

ATAVISMS . . . 41
Escape Lines . . . 41
Call Statement . . . 42
Obsolete Keywords . . . 42
Numeric Labels . . . 42
Implicit Declarations . . . 42
Computed Goto . . . 43
Goto Statement . . . 43
Dot Names . . . 43
Complex Constants . . . 44
Function Values . . . 44
Equivalence . . . 44
Minimum and Maximum Functions . . . 45

COMPILER OPTIONS . . . 45
Default Options . . . 46

Input Language Options 46
Input/Output Error Handling 46
Continuation Conventions 46
Default Formats 47
Alignments and Sizes 47
Default Input/Output Units 48
Miscellaneous Output Control Options 48
EXAMPLES 48
File Copying 48
Matrix Multiplication 49
Searching a Linked List 49
Walking a Tree 50
PORTABILITY 53
Primitives 53
Character String Copying 53
Character String Comparisons 53
DIFFERENCES BETWEEN RATFOR AND EFL 54
COMPILER 54
Current Version 54
Diagnostics 54
Quality of Fortran Produced 55
CONSTRAINTS ON EFL 57
External Names 57
Procedure Interface 57
Pointers 58
Recursion 58
Storage Allocation 58

Chapter 5

EFL: PROGRAMMING LANGUAGE

INTRODUCTION

EFL is a clean, general purpose computer language intended to encourage portable programming. It has a uniform and readable syntax and good data and control flow structuring. EFL programs can be translated into efficient Fortran code, so the EFL programmer can take advantage of the ubiquity of Fortran, the valuable libraries of software written in that language, and the portability that comes with the use of a standardized language, without suffering from Fortran's many failings as a language. It is especially useful for numeric programs. Thus, the EFL language permits the programmer to express complicated ideas in a comprehensible way, while permitting access to the power of the Fortran environment.

The name EFL originally stood for "Extended Fortran Language." The current compiler is much more than a simple preprocessor: it attempts to diagnose all syntax errors, to provide readable Fortran output, and to avoid a number of niggling restrictions.

In examples and syntax specifications, **boldface** type is used to indicate literal words and punctuation, such as **while**. Words in *italic* type indicate an item in a category, such as an *expression*. A construct surrounded by double brackets represents a list of one or more of those items, separated by commas. Thus, the notation

⟦ *item* ⟧

could refer to any of the following:

item
item, item
item, item, item

The reader should have a fair degree of familiarity with some procedural language. There will be occasional references to Ratfor and to Fortran which may be ignored if the reader is unfamiliar with those languages.

LEXICAL FORM

Character Set

The following characters are legal in an EFL program:

letters	**a b c d e f g h i j k l m** **n o p q r s t u v w x y z**
digits	**0 1 2 3 4 5 6 7 8 9**
white space	*blank tab*
quotes	' "
sharp	#
continuation	_
braces	{ }
parentheses	( )
other	, ; : . + − * / = < > & ~ ¦ $

Letter case (upper or lower) is ignored except within strings, so "**a**" and "**A**" are treated as the same character. All of the examples below are printed in lower case. An exclamation mark ("!") may be used in place of a tilde ("~"). Square brackets ("[" and "]") may be used in place of braces ("{" and "}").

Lines

EFL is a line-oriented language. Except in special cases (discussed below), the end of a line marks the end of a token and the end of a statement. The trailing portion of a line may be used for a comment. There is a mechanism for diverting input from one source file to another, so a single line in the program may be replaced by a number of lines from the other file. Diagnostic messages are labeled with the line number of the file on which they are detected.

White Space

Outside of a character string or comment, any sequence of one or more spaces or tab characters acts as a single space. Such a space terminates a token.

Comments

A comment may appear at the end of any line. It is introduced by a sharp (#) character, and continues to the end of the line. (A sharp inside of a quoted string does not mark a comment.) The sharp and succeeding characters on the line are discarded. A blank line is also a comment. Comments have no effect on execution.

Include Files

It is possible to insert the contents of a file at a point in the source text, by referencing it in a line like

include joe

No statement or comment may follow an **include** on a line. In effect, the **include** line is replaced by the lines in the named file, but diagnostics refer to the line number in the included file. **Include**s may be nested at least ten deep.

Continuation

Lines may be continued explicitly by using the underscore (_) character. If the last character of a line (after comments and trailing white space have been stripped) is an underscore, the end of a line and the initial blanks on the next line are ignored. Underscores are ignored in other contexts (except inside of quoted strings). Thus

```
1_000_000_
   000
```

equals 10^9.

There are also rules for continuing lines automatically: the end of line is ignored whenever it is obvious that the statement is not complete. To be specific, a statement is continued if the last token on a line is an operator, comma, left brace, or left parenthesis. (A statement is not continued just because of unbalanced braces or parentheses.) Some compound statements are also continued automatically; these points are noted in the sections on executable statements.

Multiple Statements on a Line

A semicolon terminates the current statement. Thus, it is possible to write more than one statement on a line. A line consisting only of a semicolon, or a semicolon following a semicolon, forms a null statement.

Tokens

A program is made up of a sequence of tokens. Each token is a sequence of characters. A blank terminates any token other than a quoted string. End of line also terminates a token unless explicit continuation (see above) is signaled by an underscore.

Identifiers

An identifier is a letter or a letter followed by letters or digits. The following is a list of the reserved words that have special meaning in EFL. They will be discussed later.

array	**exit**	**precision**
automatic	**external**	**procedure**
break	**false**	**read**
call	**field**	**readbin**
case	**for**	**real**
character	**function**	**repeat**
common	**go**	**return**
complex	**goto**	**select**
continue	**if**	**short**
debug	**implicit**	**sizeof**
default	**include**	**static**
define	**initial**	**struct**
dimension	**integer**	**subroutine**
do	**internal**	**true**
double	**lengthof**	**until**
doubleprecision	**logical**	**value**
else	**long**	**while**
end	**next**	**write**
equivalence	**option**	**writebin**

The use of these words is discussed below. These words may not be used for any other purpose.

Strings

A character string is a sequence of characters surrounded by quotation marks. If the string is bounded by single-quote marks ('), it may contain double quote marks ("), and vice versa. A quoted string may not be broken across a line boundary.

```
'hello there'
" ain't misbehavin'"
```

Integer Constants

An integer constant is a sequence of one or more digits.

```
0
57
123456
```

Floating Point Constants

A floating point constant contains a dot and/or an exponent field. An *exponent field* is a letter **d** or **e** followed by an optionally signed integer constant. If *I* and *J* are integer constants and *E* is an exponent field, then a floating constant has one of the following forms:

.I
I.
I.J
IE
I.E
.IE
I.JE

Punctuation

Certain characters are used to group or separate objects in the language. These are

```
parentheses   ( )
braces        { }
comma         ,
semicolon     ;
colon         :
end-of-line
```

The end-of-line is a token (statement separator) when the line is neither blank nor continued.

Operators

The EFL operators are written as sequences of one or more non-alphanumeric characters.


```
+   -   *   /   **
<   <=   >   >=   ==   ~=
&&  ||  &  |
+=  -=     /=  **=
&&=  ||=    &=  |=
->   .  $
```

A dot (".") is an operator when it qualifies a structure element name, but not when it acts as a decimal point in a numeric constant. There is a special mode (see " ATAVISMS") in which some of the operators may be represented by a string consisting of a dot, an identifier, and a dot (*e.g.,* **.lt.**).

Macros

EFL has a simple macro substitution facility. An identifier may be defined to be equal to a string of tokens; whenever that name appears as a token in the program, the string replaces it. A macro name is given a value in a **define** statement like

```
define count    n += 1
```

Any time the name **count** appears in the program, it is replaced by the statement

n += 1

A **define** statement must appear alone on a line; the form is

define *name rest-of-line*

Trailing comments are part of the string.

PROGRAM FORM

Files

A *file* is a sequence of lines. A file is compiled as a single unit. It may contain one or more procedures. Declarations and options that appear outside of a procedure affect the succeeding procedures on that file.

Procedures

Procedures are the largest grouping of statements in EFL. Each procedure has a name by which it is invoked. (The first procedure invoked during execution, known as the *main* procedure, has the null name.) Procedure calls and argument passing are discussed in " PROCEDURES."

Blocks

Statements may be formed into groups inside of a procedure. To describe the scope of names, it is convenient to introduce the ideas of *block* and of *nesting level.* The beginning of a program file is at nesting level zero. Any options, macro definitions, or variable declarations are also at level zero. The text immediately following a **procedure** statement is at level 1. After the declarations, a left brace marks the beginning of a new block and increases the nesting level by 1; a right brace drops the level by 1. (Braces inside declarations do not mark blocks.) (See " Blocks" under " EXECUTABLE STATEMENTS.") An **end** statement marks the end of the procedure, level 1, and the return to level 0. A name (variable or macro) that is defined at level *K* is defined throughout that block and in all deeper nested levels in which that name is not redefined or redeclared. Thus, a procedure might look like the following:

```
#  block 0
procedure george
real x
x = 2
. . .
if(x > 2)
          {           # new block
          integer x # a different variable
          do x = 1,7
                      write(,x)
          . . .
          }           # end of block
end       # end of procedure, return to block 0
```

Statements

A statement is terminated by end of line or by a semicolon. Statements are of the following types:

Option
Include
Define

Procedure
End

Declarative
Executable

The **option** statement is described in " COMPILER OPTIONS" . The **include, define,** and **end** statements have been described above; they may not be followed by another statement on a line. Each procedure begins with a **procedure** statement and finishes with an **end** statement; these are discussed in " PROCEDURES" . Declarations describe types and values of variables and procedures. Executable statements cause specific actions to be taken. A block is an example of an executable statement; it is made up of declarative and executable statements.

Labels

An executable statement may have a *label* which may be used in a branch statement. A label is an identifier followed by a colon, as in

```
        read(, x)
        if(x < 3) goto error
        . . .
error:  fatal(" bad input" )
```

DATA TYPES AND VARIABLES

EFL supports a small number of basic (scalar) types. The programmer may define objects made up of variables of basic type; other aggregates may then be defined in terms of previously defined aggregates.

Basic Types

The basic types are

logical
integer
field(*m*:*n***)**
real
complex
long real
long complex
character(*n***)**

A logical quantity may take on the two values *true* and *false*. An integer may take on any whole number value in some machine-dependent range. A field quantity is an integer restricted to a particular closed interval ([*m*:*n*]). A "real" quantity is a floating point approximation to a real or rational number. A long real is a more precise approximation to a rational. (Real quantities are represented as single precision floating point numbers; long reals are double precision floating point numbers.) A complex quantity is an approximation to a complex number, and is represented as a pair of reals. A character quantity is a fixed-length string of *n* characters.

Constants

There is a notation for a constant of each basic type.

A logical may take on the two values

true
false

An integer or field constant is a fixed point constant, optionally preceded by a plus or minus sign, as in

17
−94
+6
0

A long real ("double precision") constant is a floating point constant containing an exponent field that begins with the letter **d.** A real ("single precision") constant is any other floating point constant. A real or long real constant may be preceded by a plus or minus sign. The following are valid **real** constants:

17.3
−.4
7.9e−6 ($= 7.9\times10^{-6}$)
14e9 ($= 1.4\times10^{10}$)

The following are valid **long real** constants

7.9d−6 ($= 7.9\times10^{-6}$)
5d3

A character constant is a quoted string.

Variables

A variable is a quantity with a name and a location. At any particular time the variable may also have a value. (A variable is said to be *undefined* before it is initialized or assigned its first value, and after certain indefinite operations are performed.) Each variable has certain attributes:

Storage Class

The association of a name and a location is either transitory or permanent. Transitory association is achieved when arguments are passed to procedures. Other associations are permanent (static). (A future extension of EFL may include dynamically allocated variables.)

Scope of Names

The names of common areas are global, as are procedure names: these names may be used anywhere in the program. All other names are local to the block in which they are declared.

Precision

Floating point variables are either of normal or **long** precision. This attribute may be stated independently of the basic type.

Arrays

It is possible to declare rectangular arrays (of any dimension) of values of the same type. The index set is always a cross-product of intervals of integers. The lower and upper bounds of the intervals must be constants for arrays that are local or **common.** A formal argument array may have intervals that are of length equal to one of the other formal arguments. An element of an array is denoted by the array name followed by a parenthesized comma-separated list of integer values, each of which must lie within the corresponding interval. (The intervals may include negative numbers.) Entire arrays may be passed as procedure arguments or in input/output lists, or they may be initialized; all other array references must be to individual elements.

Structures

It is possible to define new types which are made up of elements of other types. The compound object is known as a *structure;* its constituents are called *members* of the structure. The structure may be given a name, which acts as a type name in the remaining statements within the scope of its declaration. The elements of a structure may be of any type (including previously defined structures), or they may be arrays of such objects. Entire structures may be passed to procedures or be used in input/output lists; individual elements of structures may be referenced. The uses of structures will be detailed below. The following structure might represent a symbol table:

```
struct tableentry
        {
        character(8) name
        integer hashvalue
        integer numberofelements
        field(0:1) initialized, used, set
        field(0:10) type
        }
```

EXPRESSIONS

Expressions are syntactic forms that yield a value. An expression may have any of the following forms, recursively applied:

primary
(*expression*)
unary-operator expression
expression binary-operator expression

In the following table of operators, all operators on a line have equal precedence and have higher precedence than operators on later lines. The meanings of these operators are described in " Unary Operators" and " Binary Operators" under " EXPRESSIONS" .

```
-> .
**
* /     unary + - ++ --
+ -
< <= > >= == ~=
& &&
| ||
$
= += -= *= /= **=   &= |= &&= ||=
```

Examples of expressions are

```
a<b && b<c
-(a + sin(x)) / (5+cos(x))**2
```

Primaries

Primaries are the basic elements of expressions. They include constants, variables, array elements, structure members, procedure invocations, input/output expressions, coercions, and sizes.

Constants

Constants are described in " Constants" under " DATA TYPES AND VARIABLES" .

Variables

Scalar variable names are primaries. They may appear on the left or the right side of an assignment. Unqualified names of aggregates (structures or arrays) may appear only as procedure arguments and in input/output lists.

Array Elements

An element of an array is denoted by the array name followed by a parenthesized list of subscripts, one integer value for each declared dimension:

```
a(5)
b(6, -3, 4)
```


Structure Members

A structure name followed by a dot followed by the name of a member of that structure constitutes a reference to that element. If that element is itself a structure, the reference may be further qualified.

```
a.b
x(3).y(4).z(5)
```

Procedure Invocations

A procedure is invoked by an expression of one of the forms

procedurename **( )**
procedurename **(** *expression* **)**
procedurename **(** *expression-1, ..., expression-n* **)**

The *procedurename* is either the name of a variable declared **external** or it is the name of a function known to the EFL compiler (see " Known Functions" under " PROCEDURES"), or it is the actual name of a procedure, as it appears in a **procedure** statement. If a *procedurename* is declared **external** and is an argument of the current procedure, it is associated with the procedure name passed as actual argument; otherwise it is the actual name of a procedure. Each *expression* in the above is called an *actual argument*. Examples of procedure invocations are

```
f(x)
work()
g(x, y+3, 'xx')
```

When one of these procedure invocations is to be performed, each of the actual argument expressions is first evaluated. The types, precisions, and bounds of actual and formal arguments should agree. If an actual argument is a variable name, array element, or structure member, the called procedure is permitted to use the corresponding

formal argument as the left side of an assignment or in an input list; otherwise it may only use the value. After the formal and actual arguments are associated, control is passed to the first executable statement of the procedure. When a **return** statement is executed in that procedure, or when control reaches the **end** statement of that procedure, the function value is made available as the value of the procedure invocation. The type of the value is determined by the attributes of the *procedurename* that are declared or implied in the calling procedure, which must agree with the attributes declared for the function in its procedure. In the special case of a generic function, the type of the result is also affected by the type of the argument. See " PROCEDURES" .

Input/Output Expressions

The EFL input/output syntactic forms may be used as integer primaries that have a non-zero value if an error occurs during the input or output. See " Input/Output Statements" under " EXECUTABLE STATEMENTS" .

Coercions

An expression of one precision or type may be converted to another by an expression of the form

attributes (*expression*)

At present, the only *attributes* permitted are precision and basic types. Attributes are separated by white space. An arithmetic value of one type may be coerced to any other arithmetic type; a character expression of one length may be coerced to a character expression of another length; logical expressions may not be coerced to a nonlogical type. As a special case, a quantity of **complex** or **long complex** type may be constructed from two integer or real quantities by passing two expressions (separated by a comma) in the coercion. Examples and equivalent values are

integer(5.3) = 5
long real(5) = 5.0d0
complex(5,3) = 5+3i

Most conversions are done implicitly, since most binary operators permit operands of different arithmetic types. Explicit coercions are of most use when it is necessary to convert the type of an actual argument to match that of the corresponding formal parameter in a procedure call.

Sizes

There is a notation which yields the amount of memory required to store a datum or an item of specified type:

sizeof (*leftside*)
sizeof (*attributes*)

In the first case, *leftside* can denote a variable, array, array element, or structure member. The value of **sizeof** is an integer, which gives the size in arbitrary units. If the size is needed in terms of the size of some specific unit, this can be computed by division:

sizeof(x) / sizeof(integer)

yields the size of the variable **x** in integer words.

The distance between consecutive elements of an array may not equal **sizeof** because certain data types require final padding on some machines. The **lengthof** operator gives this larger value, again in arbitrary units. The syntax is

lengthof (*leftside*)
lengthof (*attributes*)

Parentheses

An expression surrounded by parentheses is itself an expression. A parenthesized expression must be evaluated before an expression of which it is a part is evaluated.

Unary Operators

All of the unary operators in EFL are prefix operators. The result of a unary operator has the same type as its operand.

Arithmetic

Unary + has no effect. A unary − yields the negative of its operand.

The prefix operator ++ adds one to its operand. The prefix operator −− subtracts one from its operand. The value of either expression is the result of the addition or subtraction. For these two operators, the operand must be a scalar, array element, or structure member of arithmetic type. (As a side effect, the operand value is changed.)

Logical

The only logical unary operator is complement (~). This operator is defined by the equations

~ true = false
~ false = true

Binary Operators

Most EFL operators have two operands, separated by the operator. Because the character set must be limited, some of the operators are denoted by strings of two or three special characters. All binary operators except exponentiation are left associative.

Arithmetic

The binary arithmetic operators are

+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

Exponentiation is right associative: a**b**c = a**(b**c) = $a^{(b^c)}$ The operations have the conventional meanings: 8+2 = 10, 8-2 = 6, 8*2 = 16, 8/2 = 4, 8**2 = 8^2 = 64.

The type of the result of a binary operation *A op B* is determined by the types of its operands:

	Type of B				
Type of A	i	r	l r	c	l c
i	i	r	l r	c	l c
r	r	r	l r	c	l c
l r	l r	l r	l r	l c	l c
c	c	c	l c	c	l c
l c	l c	l c	l c	l c	l c

i = integer
r = real
l r = long real
c = complex
l c = long complex

If the type of an operand differs from the type of the result, the calculation is done as if the operand were first coerced to the type of the result. If both operands are integers, the result is of type integer, and is computed exactly. (Quotients are truncated toward zero, so 8/3=2.)

Logical

The two binary logical operations in EFL, **and** and **or,** are defined by the truth tables:

A	B	A and B	A or B
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

Each of these operators comes in two forms. In one form, the order of evaluation is specified. The expression

a && b

is evaluated by first evaluating **a**; if it is false then the expression is false and **b** is not evaluated; otherwise, the expression has the value of **b.** The expression

a || b

is evaluated by first evaluating **a**; if it is true then the expression is true and **b** is not evaluated; otherwise, the expression has the value of **b.** The other forms of the operators (**&** for **and** and **|** for **or**) do not imply an order of evaluation. With the latter operators, the compiler may speed up the code by evaluating the operands in any order.

Relational Operators

There are six relations between arithmetic quantities. These operators are not associative.

EFL Operator	Meaning	
<	<	less than
<=	≤	less than or equal to
==	=	equal to
~=	≠	not equal to
>	>	greater than
>=	≥	greater than or equal

Since the complex numbers are not ordered, the only relational operators that may take complex operands are == and ~= . The character collating sequence is not defined.

Assignment Operators

All of the assignment operators are right associative. The simple form of assignment is

basic-left-side = *expression*

A *basic-left-side* is a scalar variable name, array element, or structure member of basic type. This statement computes the expression on the right side, and stores that value (possibly after coercing the value to the type of the left side) in the location named by the left side. The value of the assignment expression is the value assigned to the left side after coercion.

There is also an assignment operator corresponding to each binary arithmetic and logical operator. In each case, *a op*= *b* is equivalent to *a* = *a op b*. (The operator and equal sign must not be separated by blanks.) Thus, **n+=2** adds 2 to n. The location of the left side is evaluated only once.

Dynamic Structures

EFL does not have an address (pointer, reference) type. However, there is a notation for dynamic structures,

leftside −> *structurename*

This expression is a structure with the shape implied by *structurename* but starting at the location of *leftside*. In effect, this overlays the structure template at the specified location. The *leftside* must be a variable, array, array element, or structure member. The type of the *leftside* must be one of the types in the structure declaration. An element of such a structure is denoted in the usual way using the dot operator. Thus,

place(i) −> st.elt

refers to the **elt** member of the **st** structure starting at the ith element of the array **place.**

Repetition Operator

Inside of a list, an element of the form

integer-constant-expression $ *constant-expression*

is equivalent to the appearance of the *expression* a number of times equal to the first expression. Thus,

(3, 3$4, 5)

is equivalent to

(3, 4, 4, 4, 5)

Constant Expressions

If an expression is built up out of operators (other than functions) and constants, the value of the expression is a constant, and may be used anywhere a constant is required.

DECLARATIONS

Declarations statement describe the meaning, shape, and size of named objects in the EFL language.

Syntax

A declaration statement is made up of attributes and variables. Declaration statements are of two forms:

attributes variable-list
attributes { *declarations* }

In the first case, each name in the *variable-list* has the specified attributes. In the second, each name in the declarations also has the specified attributes. A variable name may appear in more than one variable list, so long as the attributes are not contradictory. Each

name of a nonargument variable may be accompanied by an initial value specification. The *declarations* inside the braces are one or more declaration statements. Examples of declarations are

```
integer k=2

long real b(7,3)

common(cname)
    {
    integer i
    long real array(5,0:3) x, y
    character(7) ch
    }
```

Attributes

Basic Types

The following are basic types in declarations

logical
integer
field(m:n)
character(k)
real
complex

In the above, the quantities k, m, and n denote integer constant expressions with the properties $k>0$ and $n>m$.

Arrays

The dimensionality may be declared by an **array** attribute

array($b_1,...,b_n$)

Each of the b_i may either be a single integer expression or a pair of integer expressions separated by a colon. The pair of expressions form a lower and an upper bound; the single expression is an upper bound with an implied lower bound of 1. The number of dimensions is equal to n, the number of bounds. All of the integer expressions

must be constants. An exception is permitted only if all of the variables associated with an array declarator are formal arguments of the procedure; in this case, each bound must have the property that *upper-lower* +1 is equal to a formal argument of the procedure. (The compiler has limited ability to simplify expressions, but it will recognize important cases such as **(0:n−1)**. The upper bound for the last dimension (b_n) may be marked by an asterisk (*) if the size of the array is not known. The following are legal **array** attributes:

```
array(5)
array(5, 1:5, -3:0)
array(5, *)
array(0:m-1, m)
```

Structures

A structure declaration is of the form

struct *structname* { *declaration statements* }

The *structname* is optional; if it is present, it acts as if it were the name of a type in the rest of its scope. Each name that appears inside the *declarations* is a *member* of the structure, and has a special meaning when used to qualify any variable declared with the structure type. A name may appear as a member of any number of structures, and may also be the name of an ordinary variable, since a structure member name is used only in contexts where the parent type is known. The following are valid structure attributes

```
struct xx
      {
      integer a, b
      real x(5)
      }

struct { xx z(3); character(5) y }
```

The last line defines a structure containing an array of three **xx**'s and a character string.

Precision

Variables of floating point (**real** or **complex**) type may be declared to be **long** to ensure they have higher precision than ordinary floating point variables. The default precision is **short**.

Common

Certain objects called *common areas* have external scope, and may be referenced by any procedure that has a declaration for the name using a

common (*commonareaname*)

attribute. All of the variables declared with a particular **common** attribute are in the same block; the order in which they are declared is significant. Declarations for the same block in differing procedures must have the variables in the same order and with the same types, precision, and shapes, though not necessarily with the same names.

External

If a name is used as the procedure name in a procedure invocation, it is implicitly declared to have the **external** attribute. If a procedure name is to be passed as an argument, it is necessary to declare it in a statement of the form

external ⟦ *name* ⟧

If a name has the external attribute and it is a formal argument of the procedure, then it is associated with a procedure identifier passed as an actual argument at each call. If the name is not a formal argument, then that name is the actual name of a procedure, as it appears in the corresponding **procedure** statement.

Variable List

The elements of a variable list in a declaration consist of a name, an optional dimension specification, and an optional initial value specification. The name follows the usual rules. The dimension specification is the same form and meaning as the parenthesized list in an **array** attribute. The initial value specification is an equal sign (=) followed by a constant expression. If the name is an array, the right side of the equal sign may be a parenthesized list of constant expressions, or repeated elements or lists; the total number of elements in the list must not exceed the number of elements of the array, which are filled in column-major order.

The Initial Statement

An initial value may also be specified for a simple variable, array, array element, or member of a structure using a statement of the form

initial ⟦ *var* = *val* ⟧

The *var* may be a variable name, array element specification, or member of structure. The right side follows the same rules as for an initial value specification in other declaration statements.

EXECUTABLE STATEMENTS

Every useful EFL program contains executable statements, otherwise it would not do anything and would not need to be run. Statements are frequently made up of other statements. Blocks are the most obvious case, but many other forms contain statements as constituents.

To increase the legibility of EFL programs, some of the statement forms can be broken without an explicit continuation. A square (□) in the syntax represents a point where the end of a line will be ignored.

Expression Statements

Subroutine Call

A procedure invocation that returns no value is known as a subroutine call. Such an invocation is a statement. Examples are

```
work(in, out)
run()
```

Input/output statements (see "Input/Output Statements" under "EXECUTABLE STATEMENTS") resemble procedure invocations but do not yield a value. If an error occurs the program stops.

Assignment Statements

An expression that is a simple assignment (=) or a compound assignment (+= etc.) is a statement:

```
a = b
a = sin(x)/6
x *= y
```

Blocks

A block is a compound statement that acts as a statement. A block begins with a left brace, optionally followed by declarations, optionally followed by executable statements, followed by a right brace. A block may be used anywhere a statement is permitted. A block is not an expression and does not have a value. An example of a block is

```
{
integer i  # this variable is unknown
        # outside the braces
big = 0
do i = 1,n
   if(big < a(i))
        big = a(i)
}
```

Test Statements

Test statements permit execution of certain statements conditional on the truth of a predicate.

If Statement

The simplest of the test statements is the **if** statement, of form

if (*logical-expression*) □ *statement*

The logical expression is evaluated; if it is true, then the *statement* is executed.

If-Else

A more general statement is of the form

if (*logical-expression*) □ *statement-1* □
else □ *statement-2*

If the expression is **true** then *statement-1* is executed, otherwise, *statement-2* is executed. Either of the consequent statements may itself be an **if-else** so a completely nested test sequence is possible:

```
if(x<y)
   if(a<b)
      k = 1
   else
      k = 2
else
   if(a<b)
      m = 1
   else
      m = 2
```


An **else** applies to the nearest preceding un-**else**d **if**. A more common use is as a sequential test:

```
if(x==1)
   k = 1
else if(x==3 | x==5)
   k = 2
else
   k = 3
```

Select Statement

A multiway test on the value of a quantity is succinctly stated as a **select** statement, which has the general form

select(*expression*) □ *block*

Inside the block two special types of labels are recognized. A prefix of the form

case ⟦ *constant* ⟧ **:**

marks the statement to which control is passed if the expression in the select has a value equal to one of the case constants. If the expression equals none of these constants, but there is a label **default** inside the select, a branch is taken to that point; otherwise the statement following the right brace is executed. Once execution begins at a **case** or **default** label, it continues until the next **case** or **default** is encountered. The **else-if** example above is better written as

```
select(x)
   {
   case 1:
      k = 1
   case 3,5:
      k = 2
   default:
      k = 3
   }
```

Note that control does not "fall through" to the next case.

Loops

The loop forms provide the best way of repeating a statement or sequence of operations. The simplest (**while**) form is theoretically sufficient, but it is very convenient to have the more general loops available, since each expresses a mode of control that arises frequently in practice.

While Statement

This construct has the form

> **while** (*logical-expression*) □ *statement*

The expression is evaluated; if it is true, the statement is executed, and then the test is performed again. If the expression is false, execution proceeds to the next statement.

For Statement

The **for** statement is a more elaborate looping construct. It has the form

> **for** (*initial-statement* , □ *logical-expression* ,
> □ *iteration-statement*) □ *body-statement*

Except for the behavior of the **next** statement (see " Branch Statement" under " EXECUTABLE STATEMENTS"), this construct is equivalent to

```
initial-statement
while ( logical-expression )
    {
    body-statement
    iteration-statement
    }
```


This form is useful for general arithmetic iterations, and for various pointer-type operations. The sum of the integers from 1 to 100 can be computed by the fragment

```
n = 0
for(i = 1, i <= 100, i += 1)
    n += i
```

Alternatively, the computation could be done by the single statement

```
for( { n = 0 ; i = 1 } , i<=100 , { n += i ; ++i } )
    ;
```

Note that the body of the **for** loop is a null statement in this case. An example of following a linked list will be given later.

Repeat Statement

The statement

> **repeat** □ *statement*

executes the statement, then does it again, without any termination test. Obviously, a test inside the *statement* is needed to stop the loop.

Repeat ... Until Statement

The **while** loop performs a test before each iteration. The statement

> **repeat** □ *statement* □ **until** (*logical-expression*)

executes the *statement,* then evaluates the logical; if the logical is true the loop is complete; otherwise, control returns to the *statement.*

Thus, the body is always executed at least once. The **until** refers to the nearest preceding **repeat** that has not been paired with an **until.** In practice, this appears to be the least frequently used looping construct.

Do Loop

The simple arithmetic progression is a very common one in numerical applications. EFL has a special loop form for ranging over an ascending arithmetic sequence

do *variable = expression-1, expression-2, expression-3*
statement

The variable is first given the value *expression-1.* The statement is executed, then *expression-3* is added to the variable. The loop is repeated until the variable exceeds *expression-2.* If *expression-3* and the preceding comma are omitted, the increment is taken to be 1. The loop above is equivalent to

```
t2 = expression-2
t3 = expression-3
for(variable=expression-1, variable<=t2, variable+=t3)
    statement
```

(The compiler translates EFL **do** statements into Fortran DO statements, which are in turn usually compiled into excellent code.) The **do** *variable* may not be changed inside of the loop, and *expression-1* must not exceed *expression-2.* The sum of the first hundred positive integers could be computed by

```
n = 0
do i = 1, 100
        n += i
```


Branch Statements

Most of the need for branch statements in programs can be averted by using the loop and test constructs, but there are programs where they are very useful.

Goto Statement

The most general, and most dangerous, branching statement is the simple unconditional

goto *label*

After executing this statement, the next statement performed is the one following the given label. Inside of a **select** the case labels of that block may be used as labels, as in the following example:

```
select(k)
        {
        case 1:
                error(7)

        case 2:
                k = 2
                goto case 4

        case 3:
                k = 5
                goto case 4

        case 4:
                fixup(k)
                goto default

        default:
                prmsg(" ouch" )
        }
```

(If two **select** statements are nested, the case labels of the outer **select** are not accessible from the inner one.)

Break Statement

A safer statement is one which transfers control to the statement following the current **select** or loop form. A statement of this sort is almost always needed in a **repeat** loop:

```
repeat
    {
    do a computation
    if( finished )
        break
    }
```

More general forms permit controlling a branch out of more than one construct.

break 3

transfers control to the statement following the third loop and/or **select** surrounding the statement. It is possible to specify which type of construct (**for**, **while**, **repeat**, **do**, or **select**) is to be counted. The statement

break while

breaks out of the first surrounding **while** statement. Either of the statements

break 3 for
break for 3

will transfer to the statement after the third enclosing **for** loop.

Next Statement

The **next** statement causes the first surrounding loop statement to go on to the next iteration: the next operation performed is the test of a **while**, the *iteration-statement* of a **for**, the body of a **repeat**, the test of a **repeat...until**, or the increment of a **do**. Elaborations similar to those for **break** are available:

```
next
next 3
next 3 for
next for 3
```

A **next** statement ignores **select** statements.

Return

The last statement of a procedure is followed by a return of control to the caller. If it is desired to effect such a return from any other point in the procedure, a

return

statement may be executed. Inside a function procedure, the function value is specified as an argument of the statement:

return (*expression* **)**

Input/Output Statements

EFL has two input statements (**read** and **readbin**), two output statements (**write** and **writebin**), and three control statements (**endfile**, **rewind**, and **backspace**). These forms may be used either as a primary with a **integer** value or as a statement. If an exception occurs when one of these forms is used as a statement, the result is undefined but will probably be treated as a fatal error. If they are used in a context where they return a value, they return zero if no exception occurs. For the input forms, a negative value indicates end-of-file and a positive value an error. The input/output part of EFL very strongly reflects the facilities of Fortran.

Input/Output Units

Each I/O statement refers to a "unit," identified by a small positive integer. Two special units are defined by EFL, the *standard input unit* and the *standard output unit.* These particular units are assumed if no unit is specified in an I/O transmission statement.

The data on the unit are organized into *records.* These records may be read or written in a fixed sequence, and each transmission moves an integral number of records. Transmission proceeds from the first record until the *end of file.*

Binary Input/Output

The **readbin** and **writebin** statements transmit data in a machine-dependent but swift manner. The statements are of the form

> **writebin(** *unit* **,** *binary-output-list* **)**
> **readbin(** *unit* **,** *binary-input-list* **)**

Each statement moves one unformatted record between storage and the device. The *unit* is an integer expression. A *binary-output-list* is an *iolist* (see below) without any format specifiers. A *binary-input-list* is an *iolist* without format specifiers in which each of the expressions is a variable name, array element, or structure member.

Formatted Input/Output

The **read** and **write** statements transmit data in the form of lines of characters. Each statement moves one or more records (lines). Numbers are translated into decimal notation. The exact form of the lines is determined by format specifications, whether provided explicitly in the statement or implicitly. The syntax of the statements is

> **write(** *unit* **,** *formatted-output-list* **)**
> **read(** *unit* **,** *formatted-input-list* **)**

The lists are of the same form as for binary I/O, except that the lists may include format specifications. If the *unit* is omitted, the standard input or output unit is used.

Iolists

An *iolist* specifies a set of values to be written or a set of variables into which values are to be read. An *iolist* is a list of one or more *ioexpressions* of the form

> *expression*
> { *iolist* }
> *do-specification* { *iolist* }

For formatted I/O, an *ioexpression* may also have the forms

> *ioexpression* **:** *format-specifier*
> **:** *format-specifier*

A *do-specification* looks just like a **do** statement, and has a similar effect: the values in the braces are transmitted repeatedly until the **do** execution is complete.

Formats

The following are permissible *format-specifiers.* The quantities *w*, *d*, and *k* must be integer constant expressions.

i(*w***)**	integer with *w* digits
f(*w,d***)**	floating point number of *w* characters, *d* of them to the right of the decimal point.
e(*w,d***)**	floating point number of *w* characters, *d* of them to the right of the decimal point, with the exponent field marked with the letter **e**
l(*w***)**	logical field of width *w* characters, the first of which is **t** or **f** (the rest are blank on output, ignored on input) standing for **true** and **false** respectively

c	character string of width equal to the length of the datum
c(*w*)	character string of width *w*
s(*k*)	skip *k* lines
x(*k*)	skip *k* spaces
" ... "	use the characters inside the string as a Fortran format

If no format is specified for an item in a formatted input/output statement, a default form is chosen.

If an item in a list is an array name, then the entire array is transmitted as a sequence of elements, each with its own format. The elements are transmitted in column-major order, the same order used for array initializations.

Manipulation Statements

The three input/output statements

> **backspace**(*unit*)
> **rewind**(*unit*)
> **endfile**(*unit*)

look like ordinary procedure calls, but may be used either as statements or as integer expressions which yield non-zero if an error is detected. **backspace** causes the specified unit to back up, so that the next read will re-read the previous record, and the next write will over-write it. **rewind** moves the device to its beginning, so that the next input statement will read the first record. **endfile** causes the file to be marked so that the record most recently written will be the last record on the file, and any attempt to read past is an error.

PROCEDURES

Procedures are the basic unit of an EFL program, and provide the means of segmenting a program into separately compilable and named parts.

Procedures Statement

Each procedure begins with a statement of one of the forms

> **procedure**
> *attributes* **procedure** *procedurename*
> *attributes* **procedure** *procedurename* **( )**
> *attributes* **procedure** *procedurename* **(** ⟦ *name* ⟧ **)**

The first case specifies the main procedure, where execution begins. In the two other cases, the *attributes* may specify precision and type, or they may be omitted entirely. The precision and type of the procedure may be declared in an ordinary declaration statement. If no type is declared, then the procedure is called a *subroutine* and no value may be returned for it. Otherwise, the procedure is a function and a value of the declared type is returned for each call. Each *name* inside the parentheses in the last form above is called a *formal argument* of the procedure.

End Statement

Each procedure terminates with a statement

> **end**

Argument Association

When a procedure is invoked, the actual arguments are evaluated. If an actual argument is the name of a variable, an array element, or a structure member, that entity becomes associated with the formal argument, and the procedure may reference the values in the object, and assign to it. Otherwise, the value of the actual is associated with the formal argument, but the procedure may not attempt to change the value of that formal argument.

If the value of one of the arguments is changed in the procedure, it is not permitted that the corresponding actual argument be associated with another formal argument or with a **common** element that is referenced in the procedure.

Execution and Return Values

After actual and formal arguments have been associated, control passes to the first executable statement of the procedure. Control returns to the invoker either when the **end** statement of the procedure is reached or when a **return** statement is executed. If the procedure is a function (has a declared type), and a **return(***value***)** is executed, the value is coerced to the correct type and precision and returned.

Known Functions

A number of functions are known to EFL, and need not be declared. The compiler knows the types of these functions. Some of them are *generic;* i.e., they name a family of functions that differ in the types of their arguments and return values. The compiler chooses which element of the set to invoke based upon the attributes of the actual arguments.

Minimum and Maximum Functions

The generic functions are **min** and **max.** The **min** calls return the value of their smallest argument; the **max** calls return the value of their largest argument. These are the only functions that may take different numbers of arguments in different calls. If any of the arguments are **long real** then the result is **long real**. Otherwise, if any of the arguments are **real** then the result is **real;** otherwise all the arguments and the result must be **integer.** Examples are

```
min(5, x, -3.20)
max(i, z)
```

Absolute Value

The **abs** function is a generic function that returns the magnitude of its argument. For integer and real arguments the type of the result is identical to the type of the argument; for complex arguments the type of the result is the real of the same precision.

Elementary Functions

The following generic functions take arguments of **real, long real,** or **complex** type and return a result of the same type:

sin	**sine function**
cos	**cosine function**
exp	**exponential function (e^x).**
log	**natural (base e) logarithm**
log10	**common (base 10) logarithm**
sqrt	**square root function ($\sqrt{x}$).**

In addition, the following functions accept only **real** or **long real** arguments:

atan	$atan(x)=\tan^{-1}x$
atan2	$atan2(x,y)=\tan^{-1}\frac{x}{y}$

Other Generic Functions

The **sign** functions takes two arguments of identical type; **sign**$(x,y) = sgn(y)|x|$. The **mod** function yields the remainder of its first argument when divided by its second. These functions accept integer and real arguments.

ATAVISMS

Certain facilities are included in the EFL language to ease the conversion of old Fortran or Ratfor programs to EFL.

Escape Lines

In order to make use of nonstandard features of the local Fortran compiler, it is occasionally necessary to pass a particular line through to the EFL compiler output. A line that begins with a percent sign ("%") is copied through to the output, with the percent sign removed but no other change. Inside of a procedure, each escape line is treated as an executable statement. If a sequence of lines constitute a continued Fortran statement, they should be enclosed in braces.

Call Statement

A subroutine call may be preceded by the keyword **call.**

```
call joe
call work(17)
```

Obsolete Keywords

The following keywords are recognized as synonyms of EFL keywords:

Fortran	EFL
double precision	**long real**
function	**procedure**
subroutine	**procedure** *(untyped)*

Numeric Labels

Standard statement labels are identifiers. A numeric (positive integer constant) label is also permitted; the colon is optional following a numeric label.

Implicit Declarations

If a name is used but does not appear in a declaration, the EFL compiler gives a warning and assumes a declaration for it. If it is used in the context of a procedure invocation, it is assumed to be a procedure name; otherwise it is assumed to be a local variable defined at nesting level 1 in the current procedure. The assumed type is determined by the first letter of the name. The association of letters and types may be given in an **implicit** statement, with syntax

implicit (*letter-list*) *type*

where a *letter-list* is a list of individual letters or ranges (pair of letters separated by a minus sign). If no **implicit** statement appears, the following rules are assumed:

```
implicit (a–h, o–z) real
implicit (i–n) integer
```


Computed Goto

Fortran contains an indexed multi-way branch; this facility may be used in EFL by the computed GOTO:

goto (⟦ *label* ⟧), *expression*

The expression must be of type integer and be positive but be no larger than the number of labels in the list. Control is passed to the statement marked by the label whose position in the list is equal to the expression.

Goto Statement

In unconditional and computed **goto** statements, it is permissible to separate the **go** and **to** words, as in

```
go to xyz
```

Dot Names

Fortran uses a restricted character set, and represents certain operators by multi-character sequences. There is an option (**dots=on**; see " COMPILER OPTIONS") which forces the compiler to recognize the forms in the second column below:

<	**.lt.**
<=	**.le.**
>	**.gt.**
>=	**.ge.**
==	**.eq.**
~=	**.ne.**
&	**.and.**
¦	**.or.**
&&	**.andand.**
¦¦	**.oror.**
~	**.not.**
true	**.true.**
false	**.false.**

In this mode, no structure element may be named **lt, le,** etc. The readable forms in the left column are always recognized.

Complex Constants

A complex constant may be written as a parenthesized list of real quantities, such as

(1.5, 3.0)

The preferred notation is by a type coercion,

complex(1.5, 3.0)

Function Values

The preferred way to return a value from a function in EFL is the **return(***value***)** construct. However, the name of the function acts as a variable to which values may be assigned; an ordinary **return** statement returns the last value assigned to that name as the function value.

Equivalence

A statement of the form

equivalence $v_1, v_2, ..., v_n$

declares that each of the v_i starts at the same memory location. Each of the v_i may be a variable name, array element name, or structure member.

Minimum and Maximum Functions

There are a number of non-generic functions in this category, which differ in the required types of the arguments and the type of the return value. They may also have variable numbers of arguments, but all the arguments must have the same type.

Function	*Argument Type*	*Result Type*
amin0	**integer**	**real**
amin1	**real**	**real**
min0	**integer**	**integer**
min1	**real**	**integer**
dmin1	**long real**	**long real**
amax0	**integer**	**real**
amax1	**real**	**real**
max0	**integer**	**integer**
max1	**real**	**integer**
dmax1	**long real**	**long real**

COMPILER OPTIONS

A number of options can be used to control the output and to tailor it for various compilers and systems. The defaults chosen are conservative, but it is sometimes necessary to change the output to match peculiarities of the target environment.

Options are set with statements of the form

option ⟦ *opt* ⟧

where each *opt* is of one of the forms

optionname
optionname = *optionvalue*

The *optionvalue* is either a constant (numeric or string) or a name associated with that option. The two names **yes** and **no** apply to a number of options.

Default Options

Each option has a default setting. It is possible to change the whole set of defaults to those appropriate for a particular environment by using the **system** option. At present, the only valid values are **system=unix** and **system=gcos.**

Input Language Options

The **dots** option determines whether the compiler recognizes **.lt.** and similar forms. The default setting is **no.**

Input/Output Error Handling

The **ioerror** option can be given three values: **none** means that none of the I/O statements may be used in expressions, since there is no way to detect errors. The implementation of the **ibm** form uses ERR= and END= clauses. The implementation of the **fortran77** form uses IOSTAT= clauses.

Continuation Conventions

By default, continued Fortran statements are indicated by a character in column 6 (Standard Fortran). The option **continue=column1** puts an ampersand (&) in the first column of the continued lines instead.

Default Formats

If no format is specified for a datum in an iolist for a **read** or **write** statement, a default is provided. The default formats can be changed by setting certain options

Option	*Type*
iformat	integer
rformat	real
dformat	long real
zformat	complex
zdformat	long complex
lformat	logical

The associated value must be a Fortran format, such as

option rformat=f22.6

Alignments and Sizes

In order to implement **character** variables, structures, and the **sizeof** and **lengthof** operators, it is necessary to know how much space various Fortran data types require, and what boundary alignment properties they demand. The relevant options are

Fortran Type	*Size Option*	*Alignment Option*
integer	**isize**	**ialign**
real	**rsize**	**ralign**
long real	**dsize**	**dalign**
complex	**zsize**	**zalign**
logical	**lsize**	**lalign**

The sizes are given in terms of an arbitrary unit; the alignment is given in the same units. The option **charperint** gives the number of characters per **integer** variable.

Default Input/Output Units

The options **ftnin** and **ftnout** are the numbers of the standard input and output units. The default values are **ftnin=5** and **ftnout=6.**

Miscellaneous Output Control Options

Each Fortran procedure generated by the compiler will be preceded by the value of the **procheader** option.

No Hollerith strings will be passed as subroutine arguments if **hollincall=no** is specified.

The Fortran statement numbers normally start at 1 and increase by 1. It is possible to change the increment value by using the **deltastno** option.

EXAMPLES

In order to show the flavor or programming in EFL, we present a few examples. They are short, but show some of the convenience of the language.

File Copying

The following short program copies the standard input to the standard output, provided that the input is a formatted file containing lines no longer than a hundred characters.

```
procedure   # main program
character(100) line

while( read( , line) == 0 )
                    write( , line)
end
```

Since **read** returns zero until the end of file (or a read error), this program keeps reading and writing until the input is exhausted.

Matrix Multiplication

The following procedure multiplies the $m \times n$ matrix a by the $n \times p$ matrix b to give the $m \times p$ matrix c. The calculation obeys the formula $c_{ij} = \sum a_{ik} b_{kj}$.

```
procedure matmul(a,b,c, m,n,p)
integer i, j, k, m, n, p
long real a(m,n), b(n,p), c(m,p)
do i = 1,m
do j = 1,p
          {
          c(i,j) = 0
          do k = 1,n
                    c(i,j) += a(i,k) * b(k,j)
          }
end
```

Searching a Linked List

Assume we have a list of pairs of numbers (x,y). The list is stored as a linked list sorted in ascending order of x values. The following procedure searches this list for a particular value of x and returns the corresponding y value.

```
define LAST         0
define NOTFOUND          -1

integer procedure val(list, first, x)

#  list is an array of structures.
#  Each structure contains a thread index value,
#  an x, and a y value.
struct
          {
          integer nextindex
          integer x, y
          } list(*)
integer first, p, arg

for(p = first , p~=LAST && list(p).x<=x ,
   p = list(p).nextindex)
          if(list(p).x == x)
                    return( list(p).y )

return(NOTFOUND)
end
```

The search is a single **for** loop that begins with the head of the list and examines items until either the list is exhausted (p==LAST) or until it is known that the specified value is not on the list (list(p).x > x). The two tests in the conjunction must be performed in the specified order to avoid using an invalid subscript in the **list(p)** reference. Therefore, the **&&** operator is used. The next element in the chain is found by the iteration statement **p=list(p).nextindex**.

Walking a Tree

As an example of a more complicated problem, let us imagine we have an expression tree stored in a common area, and that we want to print out an infix form of the tree. Each node is either a leaf (containing a numeric value) or it is a binary operator, pointing to a

left and a right descendant. In a recursive language, such a tree walk would be implement by the following simple pseudocode:

> *if this node is a leaf*
>   *print its value*
> *otherwise*
>   *print a left parenthesis*
>   *print the left node*
>   *print the operator*
>   *print the right node*
>   *print a right parenthesis*

In a nonrecursive language like EFL, it is necessary to maintain an explicit stack to keep track of the current state of the computation. The following procedure calls a procedure **outch** to print a single character and a procedure **outval** to print a value.

```
procedure walk(first)       # print an expression tree

integer first          # index of root node
integer currentnode
integer stackdepth
common(nodes) struct
        {
        character(1) op
        integer leftp, rightp
        real val
        } tree(100)    # array of structures

struct
        {
        integer nextstate
        integer nodep
        } stackframe(100)

define NODE tree(currentnode)
define STACK         stackframe(stackdepth)

#  nextstate values
define DOWN        1
define LEFT  2
define RIGHT       3
```

```
# initialize stack with root node
stackdepth = 1
STACK.nextstate = DOWN
STACK.nodep = first

while( stackdepth > 0 )
        {
        currentnode = STACK.nodep
        select(STACK.nextstate)
                {
                case DOWN:
                        if(NODE.op == " " )  # a leaf
                                {
                                outval( NODE.val )
                                stackdepth -= 1
                                }
                        else    { # a binary operator node
                                outch( " (" )
                                STACK.nextstate = LEFT
                                stackdepth += 1
                                STACK.nextstate = DOWN
                                STACK.nodep = NODE.leftp
                                }
                case LEFT:
                        outch( NODE.op )
                        STACK.nextstate = RIGHT
                        stackdepth += 1
                        STACK.nextstate = DOWN
                        STACK.nodep = NODE.rightp
                case RIGHT:
                        outch( " )" )
                        stackdepth -= 1
                }
        }
end
```


PORTABILITY

One of the major goals of the EFL language is to make it easy to write portable programs. The output of the EFL compiler is intended to be acceptable to any Standard Fortran compiler (unless the **fortran77** option is specified).

Primitives

Certain EFL operations cannot be implemented in portable Fortran, so a few machine-dependent procedures must be provided in each environment.

Character String Copying

The subroutine **ef1asc** is called to copy one character string to another. If the target string is shorter than the source, the final characters are not copied. If the target string is longer, its end is padded with blanks. The calling sequence is

```
subroutine ef1asc(a, la, b, lb)
integer a(*), la, b(*), lb
```

and it must copy the first **lb** characters from **b** to the first **la** characters of **a**.

Character String Comparisons

The function **ef1cmc** is invoked to determine the order of two character strings. The declaration is

```
integer function ef1cmc(a, la, b, lb)
integer a(*), la, b(*), lb
```

The function returns a negative value if the string **a** of length **la** precedes the string **b** of length **lb**. It returns zero if the strings are equal, and a positive value otherwise. If the strings are of differing length, the comparison is carried out as if the end of the shorter string were padded with blanks.

DIFFERENCES BETWEEN RATFOR AND EFL

There are a number of differences between Ratfor and EFL, since EFL is a defined language while Ratfor is the union of the special control structures and the language accepted by the underlying Fortran compiler. Ratfor running over Standard Fortran is almost a subset of EFL. Most of the features described in the " ATAVISMS" are present to ease the conversion of Ratfor programs to EFL.

There are a few incompatibilities: The syntax of the **for** statement is slightly different in the two languages: the three clauses are separated by semicolons in Ratfor, but by commas in EFL. (The initial and iteration statements may be compound statements in EFL because of this change). The input/output syntax is quite different in the two languages, and there is no FORMAT statement in EFL. There are no ASSIGN or assigned GOTO statements in EFL.

The major linguistic additions are character data, factored declaration syntax, block structure, assignment and sequential test operators, generic functions, and data structures. EFL permits more general forms for expressions, and provides a more uniform syntax. (One need not worry about the Fortran/Ratfor restrictions on subscript or DO expression forms, for example.)

COMPILER

Current Version

The current version of the EFL compiler is a two-pass translator written in portable C. It implements all of the features of the language described above except for **long complex** numbers.

Diagnostics

The EFL compiler diagnoses all syntax errors. It gives the line and file name (if known) on which the error was detected. Warnings are given for variables that are used but not explicitly declared.

Quality of Fortran Produced

The Fortran produced by EFL is quite clean and readable. To the extent possible, the variable names that appear in the EFL program are used in the Fortran code. The bodies of loops and test constructs are indented. Statement numbers are consecutive. Few unneeded GOTO and CONTINUE statements are used. It is considered a compiler bug if incorrect Fortran is produced (except for escaped lines). The following is the Fortran procedure produced by the EFL compiler for the matrix multiplication example (See " EXAMPLES" .)

```
      subroutine matmul(a, b, c, m, n, p)
      integer m, n, p
      double precision a(m, n), b(n, p), c(m, p)
      integer i, j, k
      do 3 i = 1, m
         do 2 j = 1, p
            c(i, j) = 0
            do 1 k = 1, n
               c(i, j) = c(i, j)+a(i, k)*b(k, j)
  1            continue
  2         continue
  3      continue
      end
```

The following is the procedure for the tree walk:

```
        subroutine walk(first)
        integer first
        common /nodes/ tree
        integer tree(4, 100)
        real tree1(4, 100)
        integer staame(2, 100), stapth, curode
        integer const1(1)
        equivalence (tree(1,1), tree1(1,1))
        data const1(1)/4h    /
c print out an expression tree
c index of root node
c array of structures
c   nextstate values
c   initialize stack with root node
        stapth = 1
        staame(1, stapth) = 1
        staame(2, stapth) = first
    1   if (stapth .le. 0) goto  9
            curode = staame(2, stapth)
            goto  7
    2           if (tree(1, curode) .ne. const1(1)) goto 3
                    call outval(tree1(4, curode))
c a leaf
                    stapth = stapth-1
                    goto  4
    3               call outch(1h()
c a binary operator node
                    staame(1, stapth) = 2
                    stapth = stapth+1
                    staame(1, stapth) = 1
                    staame(2, stapth) = tree(2, curode)
    4           goto  8
    5           call outch(tree(1, curode))
                staame(1, stapth) = 3
                stapth = stapth+1
                staame(1, stapth) = 1
                staame(2, stapth) = tree(3, curode)
                goto  8
    6           call outch(1h))
                stapth = stapth-1
                goto  8
```



```
    7           if (staame(1, stapth) .eq. 3) goto  6
                if (staame(1, stapth) .eq. 2) goto  5
                if (staame(1, stapth) .eq. 1) goto  2
    8       continue
            goto  1
    9   continue
        end
```

CONSTRAINTS ON EFL

Although Fortran can be used to simulate any finite computation, there are realistic limits on the generality of a language that can be translated into Fortran. The design of EFL was constrained by the implementation strategy. Certain of the restrictions are petty (six character external names), but others are sweeping (lack of pointer variables). The following paragraphs describe the major limitations imposed by Fortran.

External Names

External names (procedure and COMMON block names) must be no longer than six characters in Fortran. Further, an external name is global to the entire program. Therefore, EFL can support block structure within a procedure, but it can have only one level of external name if the EFL procedures are to be compilable separately, as are Fortran procedures.

Procedure Interface

The Fortran standards, in effect, permit arguments to be passed between Fortran procedures either by reference or by copy-in/copy-out. This indeterminacy of specification shows through into EFL. A program that depends on the method of argument transmission is illegal in either language.

There are no procedure-valued variables in Fortran: a procedure name may only be passed as an argument or be invoked; it cannot be stored. Fortran (and EFL) would be noticeably simpler if a procedure variable mechanism were available.

Pointers

The most grievous problem with Fortran is its lack of a pointer-like data type. The implementation of the compiler would have been far easier if certain hard cases could have been handled by pointers. Further, the language could have been simplified considerably if pointers were accessible in Fortran. (There are several ways of simulating pointers by using subscripts, but they founder on the problems of external variables and initialization.)

Recursion

Fortran procedures are not recursive, so it was not practical to permit EFL procedures to be recursive. (Recursive procedures with arguments can be simulated only with great pain.)

Storage Allocation

The definition of Fortran does not specify the lifetime of variables. It would be possible but cumbersome to implement stack or heap storage disciplines by using COMMON blocks.

Chapter 6: LINT: C PROGRAM CHECKER

CONTENTS

GENERAL 1
Usage 1

TYPES OF MESSAGES 3
Unused Variables and Functions 3
Set/Used Information 5
Flow of Control 5
Function Values 6
Type Checking 7
Type Casts 9
Nonportable Character Use 9
Assignments of "longs" to "ints" 10
Strange Constructions 10
Old Syntax 11
Pointer Alignment 12
Multiple Uses and Side Effects 13

Chapter 6

LINT: C PROGRAM CHECKER

GENERAL

The **lint** program examines C language source programs detecting a number of bugs and obscurities. It enforces the type rules of C language more strictly than the C compiler. 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 legal. The **lint** program accepts multiple input files and library specifications and checks them for consistency.

Usage

The **lint** command has the form:

```
lint [options] files ... library-descriptors ...
```

where *options* are optional flags to control **lint** checking and messages; *files* are the files to be checked which end with **.c** or **.ln**; and *library-descriptors* are the names of libraries to be used in checking the program.

The options that are currently supported by the **lint** command are:

-a Suppress messages about assignments of long values to variables that are not long.

-b Suppress messages about break statements that cannot be reached.

-c Only check for intra-file bugs; leave external information in files suffixed with **.ln**.

-h Do not apply heuristics (which attempt to detect bugs, improve style, and reduce waste).

-n Do not check for compatibility with either the standard or the portable **lint** library.

-0 *name* Create a lint library from input files named **llib-l***name***.ln.**

-p Attempt to check portability to other dialects of C language.

-u Suppress messages about function and external variables used and not defined or defined and not used.

-v Suppress messages about unused arguments in functions.

-x Do not report variables referred to by external declarations but never used.

When more than one option is used, they should be combined into a single argument, such as, **-ab** or **-xha**.

The names of files that contain C language programs should end with the suffix **.c** which is mandatory or **lint** and the C compiler.

The **lint** program accepts certain arguments, such as:

```
-ly
```

These arguments specify libraries that contain functions used in the C language program. The source code is tested for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library arguments. These files all begin with the comment:

```
/* 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 comments can be used to specify features of the library functions.

The **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 do not result in messages. The **lint** program does not simulate a full library search algorithm and will print messages if the source files contain a redefinition of a library routine.

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 language program is run. When the **−p** option is used, another file is checked containing descriptions of the standard library routines which are expected to be portable across various machines. The **−n** option can be used to suppress all library checking.

TYPES OF MESSAGES

The following paragraphs describe the major categories of messages printed by **lint**.

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 types of errors rarely cause working programs to fail, but are a source of inefficiency and make programs harder to understand and change. Also, information about such unused variables and functions can occasionally serve to discover bugs.

The **lint** program prints messages 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 double 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 and can be discovered by using the **−x** option with the **lint** command.

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 messages 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.

Messages about unused arguments can be suppressed for one function by adding the comment:

```
/* ARGSUSED */
```

to the program before the function. This has the effect of the **-v** option for only one function. Also, the comment:

```
/* VARARGS */
```

can be used to suppress messages about variable number of arguments in calls to a function. The comment should be added before 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 with a digit giving the number of arguments which should be checked. For example:

```
/* VARARGS2 */
```

will cause only the first two arguments to be checked.

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. Conversely, many functions and variables defined elsewhere may be used. The **−u** option may be used to suppress the spurious messages which might otherwise appear.

Set/Used Information

The **lint** program attempts to detect cases where a variable is used before it is set. The **lint** program 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 print messages about some programs which are legal, but these programs would probably be considered bad on stylistic grounds. Because static and external variables are initialized to zero, no meaningful information can be discovered about their uses. The **lint** program does deal with initialized automatic variables.

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

The **lint** program attempts to detect unreachable portions of the programs which it processes. It will print messages 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 and to recognize the special cases **while(1)** and **for(;;)** as infinite loops. The **lint** program also prints messages about loops which cannot be entered at the top. Some valid

programs may have such loops which are considered to be bad style at best and bugs at worst.

The **lint** program has no way of detecting functions which are called and never returned. Thus, a call to **exit** may cause an unreachable code which **lint** does not detect. The most serious effects of this are in the determination of returned function values (see "Function Values"). If a particular place in the program cannot be reached but it is not apparent to **lint**, the comment

```
/* NOTREACHED */
```

can be added at the appropriate place. This comment will inform **lint** that a portion of the program cannot be reached.

The **lint** program will not print a message about unreachable **break** statements. Programs generated by **yacc** and especially **lex** may have hundreds of unreachable **break** statements. The **−O** option 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 that are never used. Sometimes programs incorrectly use function "values" that have never been returned. The **lint** program 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; the **lint** program 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 message 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.

On a global scale, **lint** detects cases where a function returns a value that is sometimes or never used. When the value is never used, 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.

Type Checking

The **lint** program enforces the type checking rules of C language 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

- 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 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.

If it is desired to turn off strict type checking for an expression, the comment

```
/* NOSTRICT */
```

should be added to the program immediately before the expression. This comment will prevent strict type checking for only the next line in the program.

Type Casts

The type cast feature in C language was introduced largely as an aid to producing more portable programs. Consider the assignment

```
p = 1 ;
```

where *p* is a character pointer. The **lint** program will print a message as a result of detecting this. 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 print messages 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 messages; otherwise, all legal casts are passed without comment, no matter how strange the type mixing seems to be.

Nonportable Character Use

On some systems, characters are signed quantities with a range from -128 to 127. On other C language implementations, characters take on only positive values. Thus, **lint** will print messages about certain comparisons and assignments as being illegal or nonportable. For example, the fragment

```
char c;
      ...
if( (c = getchar( )) < 0 ) ...
```

will work on one machine but will fail on machines where characters always take on positive values. The real solution is to declare *c* as an integer since **getchar** is actually returning integer values. In any case, **lint** will print the message "nonportable character comparison".

A similar issue arises with bit fields. When assignments of constant values are made to bit fields, the field may be too small to hold the value. This is especially true because on some machines bit fields are considered as signed quantities. While it may seem logical to consider that a two-bit field declared of type **int** cannot hold the value 3, the problem disappears if the bit field is declared to have type **unsigned**

Assignments of "longs" to "ints"

Bugs may arise from the assignment of **long** to an **int**, which will truncate the contents. 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**, which are truncated. Since there are a number of legitimate reasons for assigning **longs** to **ints**, the detection of these assignments is enabled by the **−a** option.

Strange Constructions

Several perfectly legal, but somewhat strange, constructions are detected by **lint**. The messages hopefully encourage better code quality, clearer style, and may even point out bugs. The **−h** option is used to **suppress** these checks. For example, in the statement

```
*p++ ;
```

the * does nothing. This provokes the message "null effect" from **lint**. The following program fragment:

```
unsigned x ;
if( x < 0 ) ...
```

results in a test that will never succeed. Similarly, the test

```
if( x > 0 ) ...
```


is equivalent to

```
if( x != 0 )
```

which may not be the intended action. The **lint** program will print the message "degenerate unsigned comparison" in these cases. If a program contains something similar to

```
if( 1 != 0 ) ...
```

lint will print the message "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 statement

```
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** option has not been used, **lint** prints messages 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 to be bad style, usually unnecessary, and frequently a bug.

Old Syntax

Several forms of older syntax are now illegal. 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 (e.g., +=, -=, ...) have no such ambiguities. To encourage the abandonment of the older forms, **lint** prints messages 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, the initialization

```
int x ( -1 ) ;
```

looks somewhat like the beginning of a function definition:

```
int x ( y ) { . . .
```


and the compiler must read past *x* in order to determine the correct meaning. 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. The **lint** program 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.

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 language on a particular machine not be unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler. 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.

The **lint** program checks for the important special case where a simple scalar variable is affected. For example, the statement

```
a[i] = b[i++];
```

will cause **lint** to print the message

```
warning: i evaluation order undefined
```

in order to call attention to this condition.

Chapter 7: UNIX IMPLEMENTATION

CONTENTS

INTRODUCTION 1

PROCESS CONTROL 1
- Process Creation and Program Execution 3
- Swapping 3
- Synchronization and Scheduling 5

I/O SYSTEM 6
- Block I/O System 7
- Character I/O System 8
  - Disk Drivers 8
  - Character Lists 9
  - Other Character Devices 10

THE FILE SYSTEM 10
- File System Implementation 12
- Mounted File Systems 14
- Other System Functions 14

Chapter 7

UNIX IMPLEMENTATION

This chapter describes the implementation of the resident UNIX kernel. The first section is a brief introduction. The second section describes how the UNIX system views processes, users, and programs. The third section describes the I/O system. The last section describes the UNIX file system.

INTRODUCTION

The UNIX kernel consists of 20,000 lines of C code and 500 lines of assembly code. The assembly code can be further broken down into 200 lines included for efficiency (they could have been written in C) and 300 lines performing hardware functions not possible in C.

This code represents 5 to 10 percent of what has been called "the UNIX operating system." The kernel is the only UNIX code that cannot be changed by a user. For this reason, the kernel should make as few real decisions as possible. The user doesn't need a million options to do the same thing. Rather, there should be one way to do a thing, but that way should be the least-common divisor of all the options that might have been provided.

PROCESS CONTROL

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

The user process may execute from a read-only text segment, shared by all processes executing the same code. There is no *functional* benefit from shared-text segments. An *efficiency* benefit is that there is no need to swap read-only segments out because the original copy on secondary memory is still current. This is a great benefit to interactive programs

that tend to be swapped while waiting for terminal input. Furthermore, if two processes are executing simultaneously from the same copy of a read-only segment, only one copy needs to reside in primary memory. This is a secondary effect, because simultaneous execution of a program is not common. It is ironic that this effect, which reduces the use of primary memory, only comes into play when there is an overabundance of primary memory—that is, when there is enough memory to keep waiting processes loaded.

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

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

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

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

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

Process Creation and Program Execution

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

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

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

Swapping

The major data associated with a process (the user data segment, the system data segment, and the text segment) are swapped to and from

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

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

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

These swapping algorithms are the most suspect in the system. With limited primary memory, these algorithms cause total swapping. This is not bad in itself, because the swapping does not impact the execution of the resident processes. However, if the swapping device must also be

used for file storage, the swapping traffic severly impacts the file system traffic. It is exactly these small systems that tend to double the use of limited disk resources.

Synchronization and Scheduling

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

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

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

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

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

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

I/O SYSTEM

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

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

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

Block I/O System

The model block I/O device consists of randomly addressed, secondary memory blocks of 512 or 1024 bytes each. The blocks are uniformly addressed 0, 1,... up to the size of the device. The block device driver emulates this model on a physical device.

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

The benefits in reduction of physical I/O of this scheme are substantial, especially considering the file system implementation. There are, however, some drawbacks. The asynchronous nature of the algorithm

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

Character I/O System

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

Disk Drivers

Disk drivers are implemented with a queue of transaction records. Each record holds a read/write flag, a primary memory address, a secondary memory address, and a transfer byte count. Swapping is accomplished by passing a record to the swapping device driver. The block I/O interface is implemented by passing such records with requests to fill and empty system buffers. The character I/O interface to the disk drivers create a transaction record that points directly into the user area. The routine that creates this record also ensures that the user is not swapped during this I/O transaction. Thus, by implementing the general

disk driver, it is possible to use the disk as a block device, a character device, and a swap device. The only really disk-specific code in normal disk drivers is the pre-sort of transactions to minimize latency for a particular device, and the actual issuing of the I/O request.

Character Lists

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

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

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

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

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

Other Character Devices

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

THE FILE SYSTEM

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

The UNIX file system is a disk data structure accessed completely through the block I/O system. As stated before, the canonical view of a "disk" is a randomly addressable array of 512-byte or 1024-byte blocks. A file system breaks the disk into four self-identifying regions. The first block (address 0) is unused by the file system. It is left aside for booting procedures. The second block (address 1) contains the so-called "super-block." This block, among other things, contains the size of the disk and the boundaries of the other regions. Next comes the ilist, a list of file definitions. Each file definition is a 64-byte structure, called an inode. The offset of a particular inode within the ilist is called its inumber. The combination of device name (major and minor numbers) and inumbers uniquely names a particular file. After the ilist, and at the end of the disk, are free storage blocks available for the contents of files.

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

An inode contains 13 disk addresses. The first 10 of these addresses point directly at the first 10 blocks of a file. If a file is larger than 10 blocks, then the eleventh address points at a block that contains the addresses of the next 128 blocks of the file. If the file is still larger than this, then the twelfth block points at up the 128 blocks, each pointing to 128 blocks of the file. Files yet larger use the thirteenth address for a "triple indirect" address. The algorithm ends here with the maximum file size of 1,082,201,087 bytes for a 512-byte file system, or 2,164,402,175 bytes for a 1024-byte file system.

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

The file system allows easy creation, easy removal, easy random accessing, and very easy space allocation. With most physical addresses confined to a small contiguous section of disk, it is also easy to dump, restore, and check the consistency of the file system. Large files suffer from indirect addressing, but the cache prevents most of the implied physical I/O without adding much execution. The space overhead properties of this scheme are quite good. For example, on one particular

file system, there are 25,000 files containing 130M bytes of date-file content. The overhead (inode, indirect blocks, and last block breakage) is about 11.5M. The directory structure supporting these files has about 1,500 directories containing 0.6M bytes of directory content and about 0.5M bytes of overhead in accessing the directories. This comes out to less than a 10 percent overhead for actual stored data. Most systems have this much overhead in padded trailing blanks alone.

File System Implementation

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

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

The user process accesses the file system with certain primitives. The most common of these are **open**, **creat**, **read**, **write**, **seek**, and **close**.

In the system data segment associated with a user, there is room for some (usually between 10 and 50) open files. This open file table consists of pointers that can be used to access corresponding inode table

entries. Associated with each of these open files is a current I/O pointer. This is a byte offset of the next read/write operation on the file. The system treats each read/write request as random with an implied seek to the I/O pointer. The user usually thinks of the file as sequential with the I/O pointer automatically counting the number of bytes that have been read/written from the file. The user may, of course, perform random I/O by setting the I/O pointer before reads/writes.

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

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

There is a type of unnamed FIFO file called a **pipe**. Implementation of **pipes** consists of implied **seeks** before each **read** or **write** to implement first-in first-out. There are also checks and synchronization to prevent

the writer from grossly outproducing the reader and to prevent the reader from overtaking the writer.

Mounted File Systems

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

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

Other System Functions

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

The UNIX kernel is an I/O multiplexer more than a complete operating system. This is as it should be. Because of this outlook, many features found in most other operating systems are missing from the UNIX kernel. For example, the UNIX kernel does not support file access methods, file disposition, file formats, file maximum size, spooling, command language, logical records, physical records, assignment of logical file names, logical file names, more than one character set, an operator's console, an operator, log-in, or log-out. Many of these things are symptoms rather than features. Many of these things are

implemented in user software using the kernel as a tool. A good example of this is the command language. Each user may have his own command language. Maintaining such code is as easy as maintaining user code. The idea of implementing "system" code with general user primitives comes directly from MULTICS.

Chapter 8: UNIX I/O SYSTEM

CONTENTS

DEVICE CLASSES 1
OVERVIEW OF I/O 2
CHARACTER DEVICE DRIVERS 3
THE BLOCK-DEVICE INTERFACE 7
BLOCK DEVICE DRIVERS 10
RAW BLOCK-DEVICE I/O 11

Chapter 8

UNIX I/O SYSTEM

This chapter is an overview of the UNIX I/O system. It guides writers of device driver routines, and therefore focuses on the environment and nature of device drivers, rather than the implementation of that part of the file system dealing with ordinary files. We assume that the reader has a good knowledge of the overall structure of the file system.

This chapter was updated and revised in 1984 by UniSoft Systems to reflect additions to the UniPlus$^+$ kernel for System V.

DEVICE CLASSES

There are two classes of device: *block* and *character*. The block interface is for devices, like disks and tapes, which can work with addressable 512-byte blocks. Ordinary magnetic tape only fits in this category because it can read any block using forward and backward spacing. Block devices can potentially contain a mounted file system. The interface to block devices is highly structured; the drivers for these devices share a great many routines as well as a pool of buffers.

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

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

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

OVERVIEW OF I/O

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

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

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

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

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

CHARACTER DEVICE DRIVERS

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

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

The *close* routine is called only when the file is closed for the last time. That is, when the last process closes the file. This means that it is not possible for the driver to maintain its own count of its users. The first

argument is the device number; the second is a flag which is non-zero if the file was open for writing in the process which closes it.

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

The *write* routine copies up to *u.u_count* characters from the user's buffer to the device, decrementing *u.u_count* for each character passed. For most drivers (which work one character at a time) the routine *cpass*( ) picks up characters from the user's buffer. Successive calls on it return the characters to be written, until *u.u_count* goes to 0 or an error occurs (when it returns -1). *Cpass* updates *u.u_count*.

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

The device's *read* routine is called under conditions similar to *write*, except that *u.u_count* is non-zero. The routine *pass(c)* returns characters to the user. It takes care of housekeeping, like *cpass*, and returns -1 when the last character specified by *u.u_count* is returned to the user. Before that, it returns 0. You can also use *iomove* as you do with *write* —the flag should be B_READ but the same cautions apply.

The "special functions" routine is invoked by the ioctl system call:

```
(*p) (dev,cmd,arg,mode)
```

where *p* is a pointer to the address of the device, *dev* is the device number, *cmd* is the user ioctl command argument, *arg* is the user argument, and *mode* is the file table flag word for the opened device

Finally, each device should have appropriate interrupt routines. When an interrupt occurs, it is turned into a C-compatible call to the device's interrupt routine. The interrupt-catching mechanism makes 16 bits of data available to the interrupt handler in *a-dev* (see <*include/sys/reg.h*>). This is conventionally used by drivers dealing with multiple similar devices to encode the minor device number.

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

```
struct clist    {
       int      c_cc;     /* character count */
struct cblock   *c_cf;    /* pointer to first */
struct cblock   *c_cl;    /* pointer to last  */
}
```

Putc places a character on the end of a queue (*c. &queue*) where *c* is the character and *queue* is a clist structure. The routine returns -1 if there is no space to put the character. Otherwise, it returns 0. *Getc* may retrieve the first character on the queue (*&queue*). This returns either the (non-negative) character or -1 (if the queue is empty).

The space for characters in queues is shared among all devices in the system. In the standard system there are only 600 character slots available. Thus, device handlers, especially write routines, must avoid gobbling up excessive numbers of characters.

The other major help available to device handlers is the sleep-wakeup mechanism. The call *sleep* (*event*, *priority*) makes the process wait (allowing other processes to run) until the *event* occurs. When the *event* occurs, the process is marked ready-to-run and the call returns when there is no process with higher *priority*.

The call *wakeup* (*event*) indicates that the *event* has happened, causing processes sleeping on the event to wake up. The *event* is arbitrary—agreed upon by the sleeper and the waker-up. By convention, it is the

address of some data area used by the driver. This guarantees that events are unique.

Processes sleeping on an event should not assume that the event has really happened. They should check that the conditions which caused them to sleep are no longer true.

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

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

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

Timeout (*func*,*arg*,*interval*) is useful if a device needs to know about real-time intervals. After *interval* sixtieths of a second, *func* is called with *arg* as argument, in the style (**func*)(*arg*). Timeouts provide real-time delays after function characters (like new-line and tab) in typewriter output and terminate an attempt to read the 201 Dataphone (dp) if there is no response within a specified number of seconds. Notice that the number of sixtieths of a second is limited to 2**31−1, since it must appear to be positive, and that only a bounded number of timeouts can be going on at once. Also, the specified *func* is called at clock-interrupt time, so it should conform to the requirements of interrupt routines in general.

THE BLOCK-DEVICE INTERFACE

Handling block devices is mediated by a collection of routines. These routines manage a set of buffers containing the images of blocks of data on the various devices. These routines assure that several processes accessing the same block of the same device in multiprogrammed fashion maintain a consistent view of the data in the block. A secondary but still important purpose is increasing the efficiency of the system by keeping in-core copies of blocks that are accessed frequently. The main data base for this mechanism is the table of buffers, *buf*. Each buffer header contains

- A pair of pointers (*b_forw*, *b_back*) maintaining a doubly-linked list of the buffers associated with a particular block device.
- A pair of pointers (*av_forw*, *av_back*) maintaining a doubly-linked list of "free" blocks (blocks which can be reallocated for another transaction). Buffers that have I/O in progress or are busy for other purposes do not appear in this list.
- The device and block number to which the buffer refers.
- A pointer to the actual storage associated with the buffer.
- A word count (the number of bytes to be transferred to or from the buffer).
- An error byte and a residual byte count to communicate information from an I/O routine to its caller.
- A flag word with bits indicating the status of the buffer. These flags are discussed below.

The interface with the rest of the system is primarily made up to seven routines. Both *bread* and *getblk* return a pointer to a buffer header for the block when given a device and a block number. The difference is that *bread* returns a buffer containing the current data for the block, while *getblk* returns a buffer containing the data in the block only if it is already in core (this is indicated by the *B_DONE* bit; see below). In either case, the buffer (and the corresponding device block) is "busy." Other processes referring to it have to wait until it becomes free. For example, *getblk* can be used when a block is about to be totally rewritten— no other process can refer to the block until the new data is placed in it.

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

The *brelse* routine makes the buffer available to other processes when given a pointer to a buffer. It is called, for example, after data is extracted following a *bread*. There are three subtly different write routines, all of which take a buffer pointer as argument, and all of which logically release the buffer for use by others and place it on the free list.

- *Bwrite* puts the buffer on the appropriate device queue, waits for the write, and sets the user's error flag, if required.
- *Bawrite* places the buffer on the device's queue, but does not wait for completion. For this reason, errors are not reflected directly to the user.
- *Bdwrite* does not start any I/O operation at all, but marks the buffer so that, if it is grabbed from the free list to contain data from some other block, the data in it will first be written out.

Use *bwrite* when you want to be sure that I/O takes place correctly, and that errors are reflected to the proper user— for example, when updating inodes. Use *bawrite* when you want more overlap (because no wait is required for I/O to finish) but when you are reasonably certain that the write is required. Use *bdwrite* when you are not sure that the write is needed at the moment. For example, *bdwrite* is called when the last byte of a *write* system call falls short of the end of a block, on the assumption that another *write* will be given soon which will re-use the same block. On the other hand, as the end of a block is passed, *bawrite* is called, since the block will probably not be accessed again soon and you want to start the writing process soon.

The routines *getblk* and *bread* dedicate the given block exclusively to the caller's use and make others wait. On the other hand, *brelse*, *bwrite*, *bawrite*, or *bdwrite* must eventually be called to free the block for use by others.

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

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

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

B_ERROR — This bit may be set to 1 when *B_DONE* is set to indicate that an I/O or other error occurred. If it is set, the *b_error* byte of the buffer header may contain an error code. If *b_error* is 0, the error code is not specified. Actually, no driver at present sets *b_error*.

B_BUSY — This bit indicates that the buffer header is dedicated to someone's exclusive use. However, the buffer remains attached to the list of blocks associated with its device. When *getblk* (or *bread*, which calls it) searches the buffer list for a given device and finds the requested block with this bit on, it sleeps until the bit clears

B_PHYS — This bit is set for raw I/O transactions.

B_WANTED — This flag is used in conjunction with the *B_BUSY* bit. Before sleeping (described above), *getblk* sets this flag. Conversely, when the block is freed and the busy bit goes down (in *brelse*) a *wakeup* is given for the block header whenever *B_WANTED* is on. This avoids having to call *wakeup* every time a buffer is freed on the chance that someone might want it.

B_AGE — This bit may be set on buffers just before releasing them. If it is on, the buffer is placed at the head of the free list, rather than at the tail. It is a performance heuristic used

when the caller decides that the same block will not soon be used again.

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

B_DELWRI This bit is set by *bdwrite* before releasing the buffer. When *getblk* (while searching for a free block) discovers the bit is 1 in a buffer it would otherwise grab, it writes block out before re-using it.

B_STALE This flag invalidates the association between the buffer and the device/block number. It is set when an error occurs or when the buffer is associated with a block on a file system that is unmounted.

BLOCK DEVICE DRIVERS

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

As with character devices, block device drivers may supply an *open* and a *close* routine, called respectively on each open and on the final close of the device. Instead of separate read and write routines, each block device driver has a *strategy* routine which is called with a pointer to a buffer header as argument. The buffer header contains a read/write flag, the core address, the block number, a byte count, and the major and minor device numbers. The strategy routine carries out the operation requested by the information in the buffer header. When the transaction is complete, the *B_DONE* (and possibly the *B_ERROR*) bits are set. If the *B_ASYNC* bit is set, *brelse* should be called; otherwise, *wakeup* is called. When the device is capable (under error-free operation) of transferring fewer words than requested, the device's word-count register should be placed in the residual count slot of the buffer header. Otherwise, the residual count should be set to 0. This is for the benefit of the magtape driver—it tells the user the actual length of the record.

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

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

A few routines are useful to block device drivers. *Iodone(bp)* arranges that the buffer to which *bp* points be released or awakened when the strategy module has finished with the buffer (either normally or after an error). (If after an error, the *B_ERROR* bit has presumably been set.)

The routine *geterror(bp)* can examine the error bit in a buffer header and reflect any error indication found there to the user. It may be called only in the non-interrupt part of a driver when I/O has completed (i.e., *B_DONE* has been set).

RAW BLOCK-DEVICE I/O

Block device drivers may be used to transfer information directly between the user's core image and the device without using buffers and in blocks as large as the caller requests. This involves setting up a character-type special file corresponding to the raw device and providing

read and *write* routines. These routines set up what is usually a private, non-shared buffer header with the appropriate information and call the device's strategy routine. Separate *open* and *close* routines may be provided but this is usually unnecessary. A special-function routine might come in handy, especially for magtape.

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